

CCfits Reference Manual

1.2

Generated by Doxygen 1.2.7

Fri Apr 11 16:05:08 2003

Contents

1 CCfits Documentation	1
2 CCfits User's Guide	3
3 CCfits Module Index	31
4 CCfits Hierarchical Index	31
5 CCfits Compound Index	33
6 CCfits Module Documentation	36
7 CCfits Namespace Documentation	37
8 CCfits Class Documentation	37

1 CCfits Documentation

1.1 Introduction

CCfits is an object oriented interface to the cfitsio library. cfitsio is a widely used library for manipulating FITS (Flexible Image Transport System) formatted files. This following documentation assumes prior knowledge of the FITS format and some knowledge of the use of the cfitsio library, which is in wide use, well developed, and available on many platforms.

Readers unfamiliar with FITS but in need of performing I/O with FITS data sets are directed to the first cfitsio manual, available at <http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html>. Information about the FITS file format and the current standard is available from <http://fits.gsfc.nasa.gov>.

The CCfits library provides an interface that allows the user to manipulate FITS format data through the high-level building blocks of FITS files and Header-Data Units (HDUs). The implementation is designed to hide the details of performing FITS I/O from the user, who will write calls that manipulate FITS objects by passing filenames and lists of strings that represent HDUs, keywords, image data and data columns. Unlike cfitsio, which typically requires several calls to access data (*e.g. open file, move to correct header, determine column containing table data, read data*) CCfits is designed to make reading data atomic. For example, it exploits internally existing optimization techniques for FITS I/O, choosing the optimal reading strategy as available [see the

cfitsio manual, Chapter 13] when data are read on initialization. Data written by CCfits will also be compliant with the FITS standard by specification of class constructors representing FITS dataset elements.

CCfits necessarily works in a fundamentally different way than cfitsio. The general pattern of usage for CCfits is: create a FITS object, which either opens a disk file or creates a new disk file, create references to existing or new HDU objects within it, and manipulated the data through the references. For files with Write access the library is designed to keep the FITS object on disk in sync with the memory copy. The additional memory copy increases the resources required by a calling program in return for some flexibility in accessing the data.

1.2 About this Manual

This document lays out the specification for the CCfits library.

The next sections document the installation procedure and the demonstration program *cookbook* which gives examples of usage with comments.

- [Installing the Package](#)
- [Getting Started](#)
- [Implementation Notes](#)
- [What's Present, What's Missing](#)

Following sections give a list of what is implemented in CCfits compared to the cfitsio library. For background information and as an example there is a section describing how CCfits is to be used in XSPEC, for which it was originally designed, which may serve to give the reader some insight into the design decisions made.

1.3 Notes

Release 1.2 introduces support for the Microsoft Windows 2000 platform with the Visual C++.net compiler. Earlier versions of Visual C++ did not have sufficient language support (templates, standard library). We have also successfully built CCfits using the gnu compilers on Mac OS X / Darwin release 6.4 (see installation page for further details). Apart from the new platform builds that justify a change in the minor release number a small number of bug fixes have been made (see the change log with the distribution).

1.4 Authors and Acknowledgements

CCfits was written as part of a re-engineering effort for the X-Ray data analysis program, XSPEC. It was designed using Rational Rose and originally implemented on

a Solaris platform by Ben Dorman (ben.dorman@gsfc.nasa.gov) to whom blame should be attached. Sandhya Bansal worked on part of the implementation and, and Paul Kunz (pfkeb@slac.stanford.edu) wrote the configuration scheme and dispensed helpful advice: both are also thanked profusely for the port to Windows2000/VC++.net. Thanks to R. Mathar (MPIA) and Patrik Jonsson (Lick Obs.) for contributing many helpful suggestions and bug reports, and ports to HP-UX and AIX respectively.

Suggestions and bug reports are welcome, as are offers to fill out parts of the implementation that are missing. We are also interested in knowing which parts of cfitsio that are not currently supported should be the highest priority for future extensions.

2 CCfits User's Guide

2.1 User Guide Contents

Installing the Package	4
Implementation Notes	8
Xspec and CCfits	9
Getting Started	10
Writing The Primary Image	13
Creating and Writing to an Ascii Table Extension	16
Creating and Writing to a Binary Table Extension	19
Copying an Extension between Files	24
Selecting Table Data	25
Reading Header information from a HDU	26
Reading an Image	27
Reading a Table Extension	28
What's Present, What's Missing	29
Todo List	31

2.2 Installing the Package

2.2.1 Platforms

CCfits was originally developed and tested primarily on Solaris using the SunPro CC compiler 5.1 [Sun Workshop/Forte 6.0] *prior versions of SunPro CC will not compile CCfits as they do not support templated member functions.*

CCfits has also been built and tested on Linux on the RedHat 8.0, 7.x, 6.2 distributions using g++ 2.95.3 and 3.0, and on OSF1 v. 4.0D with g++ /2.95.3. Windows 2000/VC++.net, and Mac OSX (g++ 3.1) CCfits will also build with gcc 2.95.2.

In summary, this release has been tested on the following platforms:

Platform	Compiler	Optimization
Solaris 7,8	SunPro CC 5.1, 5.3	-g and -fast
Linux/RedHat 8.0,7.x,6.2	g++ /2.95.3, 3.0.x, 3.1, 3.2.x	-g -O2
Windows 2000	VC++.net	see below
Mac OS X / Darwin 6.4	g++ 3.1	not gcc 3.2.2 (see below)

Previous versions have also been tested on the following platforms. This release has not been exercised on these, but should still work (please let us know if not!).

Platform	Compiler	Optimization
OSF1/4.0D	g++ /2.95.3	-g -O2
HP-UX 10.20	g++ /2.95.x	-g -O2
AIX	kcc	(use Makefile.kcc)

N.B. For exception support, the g++ compiler needs to be built with threads enabled (`-enable-threads` or similar option). While this is supposed to be the default on some platforms, it is wise to ensure that it is turned on by explicitly specifying it when the compiler is (re)-built. If this is not done, CCfits may dump core whenever the first exception is thrown in your code.

Not yet supported are: Solaris with gcc compilers (builds, standard library related run time error in test program).

To build and install CCfits from source code on a UNIX-like (e.g. UNIX, Linux, or Cygwin) platform, take the following steps.

2.2.2 Autogen

If you obtained the source from the CVS repository, you need to do this first step. If you obtained the source from a gzipped tar file, skip to the step 2.

If you want to do a build with a compiler other than GCC, then you need to turn off automatic dependency generation which only works with GCC. Edit the Makefile.am file by adding ‘no-dependencies’ to the line

```
AUTOMAKE_OPTIONS = foreign
```

If you do not do this, some options will be passed to compiler that it might not understand. At least with the Sun C++ compiler, these options are ignored but lead to a warning message for each file that is compiled. Other compilers may treat it as an error.

Type in the following in the toplevel source directory:

```
> ./autogen
```

autogen is a UNIX shell script that runs a series of commands in a particular order. It requires automake, autoconf and libtool to be installed on your system.

Note that you will obtain the following message after running 'autogen':

```
configure.in: 36: required file ‘config/ltconfig’ not found
```

Disregard this message. It is not an error.

2.2.3 Configure

By default, the g++ compiler and linker will be used. If you want to compile and link with a different compiler and linker, you can set some environment variable before running the configure script. For example, to use Sun's C++ compiler, do the following if you are a csh user:

```
> setenv CXX CC
```

and the equivalent if you are not. You can set the absolute path to the compiler you want to use if necessary.

By default, the configuration sets the compiler optimization flags environment variable to -g -O2 for g++ and -g for other compilers. If you wish to override this (for example, to use -O4 or -fast for SunPro CC or to switch off the optimization on g++), set this variable before running configure.

CCfits requires that the 'cfitsio' package, version 2.4 or later, is available on your system. The configure script that you will run takes an option to specify the location of the cfitsio package. The current version has been tested with cfitsio v2.410.

If the 'cfitsio' package is installed in a directory consisting of a 'lib' subdirectory with the libcfitsio.a file and a 'include' subdirectory with the fitsio.h file, then you can run the configure script with a single option. For example, if the cfitsio package is installed in this fashion in '/usr/local/ftools' then the configure script option will be...

```
-with-cfitsio=/usr/local/ftools
```

If the 'cfitsio' package is not installed in the above manner, then you need to run the configure script with two options, one to specify the include directory and the other to specify the library directory. For example, if the cfitsio package was built in '/home/user/cfitsio' then the two options will be...

```
-with-cfitsio-include=/home/user/cfitsio -with-cfitsio-libdir=/home/user/cfitsio
```

You have the option of carrying out the build in a separate directory from the source directory or in the same directory as the source. In either case, you need to run the ‘configure’ script in the directory where the build will occur. So for example, if building in the source directory with the ‘cfitsio’ directory in ‘/usr/local/ftools’ then the configure command should be issued like this...

```
> ./configure --with-cfitsio=/usr/local/ftools
```

If you do the build in separate directory from the source, you may need to issue the configure command something like this...

```
> ../CCfits/configure --with-cfitsio=/usr/local/ftools
```

The configure script will create the Makefile with the path to the compiler you choose (or GCC by default), and the path to cfitsio package. The configure script has other options, just as the install location. To see this options type

```
> ./configure --help
```

2.2.4 Build

Building the C++ shared library will be done automatically by running make without arguments like thus... *On platforms (e.g. OSF) that have a native make program, use of gnu make (gmake) only is supported. On linux, make and gmake are usually synonymous.*

```
> gmake
```

This will also build the cookbook executable.

To build on Solaris using the compiler’s optimizer, we recommend

```
> gmake CXXFLAGS=-fast
```

or

```
> gmake CXXFLAGS=-O4
```

[the -fast option will select the best optimization for the particular machine you are compiling on, which may not be the best option if your network is heterogeneous, with SparcStations and Ultra-based workstations of various vintages].

Finally, type in the following to generate the html documentation...

```
> gmake docs
```

This will create a directory in your current directory called /html, and all of the html documentation will be created in this directory. The tool ‘Doxygen’ (<http://www.doxygen.org>) is required to create the documentation. Executing this won’t quite give you a manual identical to the one you are reading. However, this option will allow the reader familiar with doxygen to gain additional information about the implementation by playing with the Doxyfile. This may be of utility to users interested in contributing to the library.

2.2.5 Install

To install, type...

```
> make install
```

The default install location will be /usr/local/lib for the library and /usr/local/include for the header files. You can change this with the –prefix option when you configure, or with something like...

```
> make DESTDIR=/usr/local/ftools install
```

2.2.6 Mac OS X / Darwin

CCfits 1.2 will compile on Mac OS X / Darwin on kernel 6.4 with gcc 3.1. We have not tested the release with any other kernel version as yet. The procedure is the same as for Linux (above). However since at time of release there appears to be a problem linking using with the current g++ 3.2.2 version, regrettably it cannot be supported.

2.2.7 Windows/Visual C++

Compiling CCfits with MS VC++ requires VC++ 7.0 or later. This is the compiler that comes with Visual Studio.NET. Earlier versions of the compiler has too many defects in the area of instantiating templates.

Take the following steps.

1. Compile the C++ code. Open the vs.net/CCfits/CCfits.sln file with Visual Studio.NET. The includes paths have been set to find the cfitsio build directory at the same level as the CCfits directory. If this is not the case, use Visual Studio.NET to edit the include paths and extra library paths to where you have cfitsio installed.

Next, just use the build icon or the build menu item.

To build the test program, cookbook, use the vs.net/cookbook.cookbook.sln file

2.3 Implementation Notes

This section comments on some of the design decisions for CCfits. We note the role of cfitsio in CCfits as the underlying "engine," the use of the C++ standard library. We also explain some of the choices made for standard library containers in the implementation - all of which is hidden from the user [as it should be].

2.3.1 CCfits wraps, not replaces, cfitsio.

Most importantly, the library wraps rather than replaces the use of cfitsio library; it does not perform direct disk I/O. The scheme is designed to retain the well-developed facilities of cfitsio (in particular, the extended file syntax), and make them available to C++ programmers in an OO framework. Some efficiency is lost over a 'pure' C++ FITS library, since the internal C implementation of many functions requires processing if blocks or switch statements that could be recoded in C++ using templates. However, we believe that the current version strikes a resonable compromise between developer time, utility and efficiency.

2.3.2 ANSI C++ and the Standard C++ Library

The implementation of CCfits uses the C++ Standard Library containers and algorithms [also referred to as the Standard Template Library, (STL)] and exception handling. Here is a summary of the rationale behind the implementation decisions made.

- HDUs are contained within a FITS object using a `std::multimap<string, HDU*>` object.
 1. The map object constructs new array members on first reference
 2. Objects stored in the map are sorted on entry and retrieved efficiently using binary search techniques.
 3. The pointer-to-HDU implementation allows for polymorphism: one set of operations will process all HDU objects within the FITS file
 4. String objects (`char*`) are represented by the `std::string` class, which has a rich public interface of search and manipulation facilities.
- Scalar column data [one entry per cell] are implemented using `std::vector<T>` objects.
- Vector column data [multiple and either fixed or variable numbers of entries per cell] are implemented using `std::vector<std::valarray <T> >` objects. The `std::valarray` template is intended for optimized numeric processing. valarrays are have the following desirable features:
 1. they are dynamic, but designed to be allocated in full on construction rather than dynamic resizing during operation: this is, what is usually needed in FITS files.
 2. They have built-in vectorized transcendental functions (e.g. `std::valarray<T> sin(const std::valarray<T> &)`).
 3. They provide `std::valarray<T> apply(T f(const T&))` operation, to apply a function `f` to each element
 4. They provide slicing operations [see the "Getting Started" section for a simple example].

- Exceptions are provided in for by a FitsException hierarchy, which prints out messages on errors and returns control to wherever the exception is caught. Non-zero status values returned by cfitsio are caught by subclass FitsError, which prints the string corresponding to an input status flag. FitsException's other subclasses are thrown on array bounds errors and other programming errors. Rare [we hope] errors that indicate programming flaws in the library throw FitsFatal errors that suggest that the user report the bug.

2.4 Xspec and CCfits

This section is provided for background. Users of CCfits need not read it except to understand how the library was conceived and therefore what its strengths and weaknesses are likely to be in this initial release.

2.4.1 About XSPEC

XSPEC is a general-purpose, multi-mission X-Ray spectral data analysis program which fits data with theoretical models by convolving those models through the instrumental responses. In XSPEC 11.x and all prior versions that use FITS format data, each individual data file format that is supported can have up to 4 ancillary files. That is, for each data file, there can be a response, correction, background and auxiliary response (efficiency) file. Additionally there are table models that read FITS format data. All told, therefore, much duplicated procedural code for reading FITS data can be eliminated by use of the greater encapsulation provided by CCfits. XSPEC's primary need is to read FITS floating point Binary Tables. XSPEC also creates simulated data by convolving users' models with detector responses, so it also has a need for writing tabular data. Images are not used in XSPEC. We have provided a support for image operations which has undergone limited testing compared to the reading interface for table extensions.

2.4.2 New Data Formats

New formats to be read in XSPEC that are specialized for a particular satellite mission can be supported almost trivially by adding new classes that read data specified with different FITS format files. A single constructor call specifying the required columns and keys is all that is needed to read FITS files, rather than a set of individual cfitsio calls. The library is designed to encourage the “resource acquisition is initialization” model of resource management; CCfits will perform more efficiently if data are read on construction.

2.4.3 Modularity

Third, in an object oriented design, it is possible to make a program only loosely dependent on current implementation assumptions. In XSPEC, data are read as SF and FITS format (of three different varieties) and the user interface is written in tcl/tk. Both of these assumptions could be changed over the future life of the program. Thus the design of XSPEC, and any similar program, consists of defining an abstract DataSet class which has a subclass that uses FITS data. The virtual functions that support reading and writing can easily be overloaded by alternatives to FITS. Thus, the class library specified here fits in with the need for modularity in design.

2.5 Getting Started

The program cookbook.cxx, analogous to the cookbook.c program supplied with cfitsio, was generated to test the correct functioning of the parts of the library and to provide a demonstration of its usage.

The code for cookbook is reproduced here with commentary as worked example of the usage of the library.

2.6 Driver Program

```
// The CCfits headers are expected to be installed in a subdirectory of
// the include path.

// The <CCfits> header file contains all that is necessary to use both the CCfits
// library and the cfitsio library (for example, it includes fitsio.h) thus making
// all of cfitsio's macro definitions available.

#ifndef HAVE_CONFIG_H
#include "config.h"
#endif

// this includes 12 of the CCfits headers and will support all CCfits operations.
// the installed location of the library headers is $(ROOT)/include/CCfits

// to use the library either add -I$(ROOT)/include/CCfits or #include <CCfits/CCfits>
// in the compilation target.

#include <CCfits>

#include <cmath>
// The library is enclosed in a namespace.
```

```

using namespace CCfits;

int main();
int writeImage();
int writeAscii();
int writeBinary();
int copyHDU();
int selectRows();
int readHeader();
int readImage();
int readTable();

int main()
{
    FITS::setVerboseMode(true);

    try
    {
        if (!writeImage()) std::cerr << " writeImage() \n";
        if (!writeAscii()) std::cerr << " writeAscii() \n";
        if (!writeBinary()) std::cerr << " writeBinary() \n";
        if (!copyHDU()) std::cerr << " copyHDU() \n";
        if (!readHeader()) std::cerr << " readHeader() \n";
        if (!readImage()) std::cerr << " readImage() \n";
        if (!readTable()) std::cerr << " readTable() \n";
        if (!selectRows()) std::cerr << " selectRows() \n";

    }
    catch (FitsException&)
    // will catch all exceptions thrown by CCfits, including errors
    // found by cfitsio (status != 0)
    {
        std::cerr << " Fits Exception Thrown by test function \n";
    }
    return 0;
}

```

The simple driver program illustrates the setting of verbose mode for the library, which makes all internal exceptions visible to the programmer. This is primarily for debugging purposes; exceptions are in some cases used to transfer control in common circumstances (e.g. testing whether a file should be created or appended to in write operations). Most of the exceptions will not produce a message unless this flag is set.

Nearly all of the exceptions thrown by CCfits are derived from `FitsException`, which is caught by reference in the above example. This includes all nonzero status codes

returned by cfitsio by the following construct (recall that in the `cfitsio library` nearly all functions return a non-zero status code on error, and have a final argument status of type int):

```
if ( [cfitsio call](args,...,&status)) throw FitsError(status);
```

FitsError, derived from FitsException, uses a cfitsio library call to convert the status code to a string message.

The few exceptions that are not derived from FitsException indicate fatal conditions implying bugs in the library. These print a message suggesting the user contact [HEASARC](#) to report the problem.

Note also the lack of statements for closing files in any of the following routines, The destructor (dtor) for the FITS object does this when it falls out of scope. A call

`FITS::destroy() throw()`

is provided for closing files explicitly; `destroy()` is also responsible for cleaning up the FITS object and deallocating its resources.

When the data are being read instead of written, the user is expected to copy the data into other program variables [rather than use references to the data contained in the FITS object].

The routines in this program test the following functionality:

`writeImage()` [Writing Primary Images and Image Extensions](#)

`writeAscii()` `ascii`

`writeBinary()` [Creating and Writing to a Binary Table Extension](#)

`copyHDU()` [Copying an Extension between Files](#)

`selectRows()` [Selecting Table Data](#)

`readHeader()` [Reading Header information from a HDU](#)

`readImage()` [Reading an Image](#)

`readTable()` [Reading a Table Extension](#)

2.7 Writing Primary Images and Image Extensions

This section of the code demonstrates creation of images. Because every fits file must have a PHDU element, all the FITS constructors (ctors) instantiate a PHDU object. In the case of a new file, the default is to establish an empty HDU with BITPIX = 8 (BYTE_IMG). A *current limitation of the code is that the data type of the PHDU*

cannot be replaced after the FITS file is created. Arguments to the FITS ctors allow the specification of the data type and the number of axes and their lengths. An image extension of type float is also written by calls in between the writes to the primary header demonstrating switch between HDUs during writes.

Note that in the example below data of type *float* is written to an image of type *unsigned int*, demonstrating both implicit type conversion and the cfitsio extension to unsigned data.

User keywords can be added to the PHDU after successful construction and these will both be accessible as container contents in the in-memory FITS object as well as being written to disk by cfitsio.

Images are represented by the standard library valarray template class which supports vectorized operations on numeric arrays (e.g. taking the square root of an array) and slicing techniques.

The code below also illustrates use of C++ standard library algorithms, and the facilities provided by the std::valarray class.

```
int writeImage()
{
    // Create a FITS primary array containing a 2-D image
    // declare axis arrays.
    long naxis      = 2;
    long naxes[2] = { 300, 200 };

    // declare auto-pointer to FITS at function scope. Ensures no resources
    // leaked if something fails in dynamic allocation.
    std::auto_ptr<FITS> pFits(0);

    try
    {
        // overwrite existing file if the file already exists.

        const std::string fileName("!atestfil.fit");

        // Create a new FITS object, specifying the data type and axes for the primary
        // image. Simultaneously create the corresponding file.

        // this image is unsigned short data, demonstrating the cfitsio extension
        // to the FITS standard.

        pFits.reset( new FITS(fileName, USHORT_IMG, naxis, naxes) );
    }
    catch (FITS::CantCreate)
    {
        // ... or not, as the case may be.
        return -1;
    }
}
```

```
// references for clarity.

long& vectorLength = naxes[0];
long& numberOfRows = naxes[1];
long nelements(1);

// Find the total size of the array.
// this is a little fancier than necessary ( It's only
// calculating naxes[0]*naxes[1]) but it demonstrates use of the
// C++ standard library accumulate algorithm.

nelements = std::accumulate(&naxes[0],&naxes[naxis],1,std::multiplies<long>());

// create a new image extension with a 300x300 array containing
// float data.

std::vector<long> extAx(2,300);
string newName ("NEW-EXTENSION");
ExtHdu* imageExt = pFits->addImage(newName,FLOAT_IMG,extAx);

// create a dummy row with a ramp. Create an array and copy the row to
// row-sized slices. [also demonstrates the use of valarray
// slices].
// also demonstrate implicit type conversion when writing to the image:
// input array will be of type float.

std::valarray<int> row(vectorLength);
for (long j = 0; j < vectorLength; ++j) row[j] = j;
std::valarray<int> array(nelements);
for (int i = 0; i < numberOfRows; ++i)
{
    array[std::slice(vectorLength*static_cast<int>(i),vectorLength,1)] = row + i;
}

// create some data for the image extension.
long extElements = std::accumulate(extAx.begin(),extAx.end(),1,std::multiplies<long>());
std::valarray<float> ranData(extElements);
const float PIBY (M_PI/150.);
for ( int jj = 0 ; jj < extElements ; ++jj)
{
    float arg = PIBY*jj;
    ranData[jj] = std::cos(arg);
}

long fpixel(1);

// write the image extension data: also demonstrates switching
// between
// HDUs.
imageExt->write(fpixel,extElements,ranData);
```

```

//add two keys to the primary header, one long, one complex.

long exposure(1500);
std::complex<float> omega(std::cos(2*M_PI/3.),std::sin(2*M_PI/3));
pFits->pHDU().addKey("EXPOSURE", exposure,"Total Exposure Time");
pFits->pHDU().addKey("OMEGA",omega," Complex cube root of 1 ");

// The function PHDU& FITS::pHDU() returns a reference to the ob-
ject representing
// the primary HDU; PHDU::write( <args> ) is then used to write the data.

pFits->pHDU().write(fpixel,nelements,array);

// PHDU's friend ostream operator. Doesn't print the entire ar-
ray, just the
// required & user keywords, and is provided largely for test-
ing purposes [see
// readImage() for an example of how to output the image ar-
ray to a stream].

std::cout << pFits->pHDU() << std::endl;

return 0;
}

```

2.8 Creating and Writing to an Ascii Table Extension

In this section of the program we create a new Table extension of type AsciiTbl, and write three columns with 6 rows. Then we add another copy of the data two rows down (starting from row 3) thus overwriting values and creating new rows. We test the use of null values, and writing a date string. Implicit data conversion, as illustrated for images above, is supported. However, writing numeric data as character data, supported by cfitsio, is *not* supported by CCfits.

Note the basic pattern of CCfits operations: they are performed on an object of type FITS. Access to HDU extension is provided by FITS:: member functions that return references or pointers to objects representing HDUs. Extension are never created directly (all extension ctors are protected), but only through the functions FITS::addTable and FITS::addImage which add extensions to an existing FITS object, performing the necessary cfitsio calls.

The FITS::addTable function takes as one of its last arguments a HDU Type parameter, which needs to be AsciiTbl or BinTbl. The default is to create a BinTable (see next function).

Similarly, access to column data is provided through the functions *ExtHDU::Column*, which return references to columns specified by name or index number - see the documentation for the class ExtHDU for details.

addTable returns a pointer to Table, which is the abstract immediate superclass of the concrete classes AsciiTable and BinTable, whereas addImage returns a pointer to ExtHDU, which is the abstract base class of all FITS extensions. These base classes implement the public interface necessary to avoid the user of the library needing to downcast to a concrete type.

```

int writeAscii ()

//*****
// Create an ASCII Table extension containing 3 columns and 6 rows *
//*****
{
    // declare auto-pointer to FITS at function scope. Ensures no resources
    // leaked if something fails in dynamic allocation.
    std::auto_ptr<FITS> pFits(0);

    try
    {

        const std::string fileName("atestfil.fit");

        // append the new extension to file created in previous function call.
        // CCfits writing constructor.

        // if this had been a new file, then the following code would create
        // a dummy primary array with BITPIX=8 and NAXIS=0.

        pFits.reset( new FITS(fileName,Write) );
    }
    catch (CCfits::FITS::CantOpen)
    {
        // ... or not, as the case may be.
        return -1;
    }

    unsigned long rows(6);
    string hduName("PLANETS_ASCII");
    std::vector<string> colName(3,"");
    std::vector<string> colForm(3,"");
    std::vector<string> colUnit(3,"");

    colName[0] = "Planet";
    colName[1] = "Diameter";
    colName[2] = "Density";

    colForm[0] = "a8";
    colForm[1] = "i6";
    colForm[2] = "f4.2";
}

```

```

colUnit[0] = "";
colUnit[1] = "km";
colUnit[2] = "g/cm^‐3";

std::vector<string> planets(rows);

const char *planet[] = {"Mercury", "Venus", "Earth",
                       "Mars", "Jupiter", "Saturn"};
const char *mnemoy[] = {"Many", "Volcanoes", "Erupt",
                       "Mulberry", "Jam", "Sandwiches", "Under",
                       "Normal", "Pressure"};

long diameter[] = {4880,     12112,     12742,     6800,    143000,    121000};
float density[] = { 5.1,      5.3,      5.52,     3.94,     1.33,      0.69};

// append a new ASCII table to the fits file. Note that the user
// cannot call the Ascii or Bin Table constructors directly as they
// are protected.

Table* newTable = pFits->addTable(hduName,rows,colName,colForm,colUnit,AsciiTbl);

for (size_t j = 0; j < rows; ++j) planets[j] = string(planet[j]);

// Table::column(const std::string& name) returns a reference
// to a Column object

try
{
    newTable->column(colName[0]).write(planets,1);
    newTable->column(colName[1]).write(diameter,rows,1);
    newTable->column(colName[2]).write(density,rows,1);

}
catch (FitsException&)
{
    // ExtHDU::column could in principle throw a NoSuchColumn exception,
    // or some other fits error may ensue.
    std::cerr << " Error in writing to columns -
check e.g. that columns of specified name "
               << " exist in the extension \n";
}

// FITSUtil::auto_array_ptr<T> is provided to counter re-
source leaks that
// may arise from C-arrays. It is a std::auto_ptr<T> ana-
log that calls
// delete[] instead of delete.

FITSUtil::auto_array_ptr<long> pDiameter(new long[rows]);
FITSUtil::auto_array_ptr<float> pDensity(new float[rows]);
long* Cdiameter(pDiameter.get());

```

```
float* Cdensity(pDensity.get());

Cdiameter[0] = 4880; Cdiameter[1] = 12112; Cdiameter[2] = 12742; Cdiameter[3] = 6800;
Cdiameter[4] = 143000; Cdiameter[5] = 121000;

Cdensity[0] = 5.1; Cdensity[1] = 5.3; Cdensity[2] = 5.52; Cdensity[3] = 3.94; Cdensity[4] = 1.33;
Cdensity[5] = 0.69;

// this << operator outputs everything that has been read.

std::cout << *newTable << std::endl;

pFits->pHDU().addKey("NEWVALUE", 42, " Test of adding keyword to different extension");

// demonstrate increasing number of rows and null values.
long ignoreVal(12112);
long nullNumber(-999);
try
{
    // add a TNULLn value to column 2.
    newTable->column(colName[1]).addNullValue(nullNumber);
    // test that writing new data properly expands the number of rows
    // in both the file].write(planets,rows-3);
    newTable->column(colName[2]).write(density,rows,rows-3);
    // test the undefined value functionality. Undefineds are replaced on
    // disk but not in the memory copy.
    newTable->column(colName[1]).write(diameter,rows,rows-3,&ignoreVal);
}
catch (FitsException&)
{
    // this time we're going to ignore problems in these operations
}

// output header information to check that everything we did so far
// hasn't corrupted the file.

std::cout << pFits->pHDU() << std::endl;

std::vector<string> mnemon(9);
for (size_t j = 0; j < 9; ++j) mnemon[j] = string(mnemoy[j]);

// Add a new column of string type to the Table.
// type, columnName, width, units. [optional - decimals, column number]
// decimals is only relevant for floating-point data in ascii columns.
```

```

newTable->addColumn(Tstring,"Mnemonic",10," words ");
newTable->column("Mnemonic").write(mnemon,1);

// write the data string.
newTable->writeDate();

// and see if it all worked right.
std::cout << *newTable << std::endl;

return 0;
}

```

2.9 Creating and Writing to a Binary Table Extension

The Binary Table interface is more complex because there is an additional parameter, the vector size of each ‘cell’ in the table, the need to support variable width columns, and the desirability of supporting the input of data in various formats.

The interface supports writing to vector tables the following data structures: C-arrays (`T*`), `std::vector<T>` objects, `std::valarray<T>` objects, and `std::vector<valarray<T>>`. The last of these is the internal representation of the data.

The function below exercises the following functionality:

- Create a BinTable extension
- Write vector rows to the table
- Insert table rows
- Write complex data to both scalar and vector columns.
- Insert Table columns
- Delete Table rows
- Write HISTORY and COMMENT cards to the Table

```

int writeBinary ()

//*****
// Create a BINARY table extension and write and manipulate vec-
tor rows
//*****
{
    std::auto_ptr<FITS> pFits(0);

    try
    {
        const std::string fileName("atestfil.fit");
        pFits.reset( new FITS(fileName,Write) );
    }
    catch (CCfits::FITS::CantOpen)

```

```

{
    return -1;
}

unsigned long rows(3);
string hduName("TABLE_BINARY");
std::vector<string> colName(7, "");
std::vector<string> colForm(7, "");
std::vector<string> colUnit(7, "");

colName[0] = "numbers";
colName[1] = "sequences";
colName[2] = "powers";
colName[3] = "big-integers";
colName[4] = "dcomplex-roots";
colName[5] = "fcomplex-roots";
colName[6] = "scalar-complex";

colForm[0] = "8A";
colForm[1] = "20J";
colForm[2] = "20D";
colForm[3] = "20V";
colForm[4] = "20M";
colForm[5] = "20C";
colForm[6] = "1M";

colUnit[0] = "magnets";
colUnit[1] = "bulbs";
colUnit[2] = "batteries";
colUnit[3] = "mulberries";
colUnit[4] = "";
colUnit[5] = "";
colUnit[6] = "pico boo";

std::vector<string> numbers(rows);

const string num("NUMBER-");
for (size_t j = 0; j < rows; ++j)
{
#ifdef HAVE_OSTRSTREAM
    std::ostrstream pStr;
#else
    std::ostringstream pStr;
#endif
    pStr << num << j+1;
    numbers[j] = string(pStr.str());
}

const size_t OFFSET(20);

// write operations take in data as valarray<T>, vector<T> , and
// vector<valarray<T> >, and T* C-arrays. Create arrays to exercise the C++

```

```

// containers. Check complex I/O for both float and double complex types.

std::valarray<long> sequence(60);
std::vector<long> sequenceVector(60);
std::vector<std::valarray<long>> sequenceVV(3);

for (size_t j = 0; j < rows; ++j)
{
    sequence[OFFSET*j] = 1 + j;
    sequence[OFFSET*j+1] = 1 + j;
    sequenceVector[OFFSET*j] = sequence[OFFSET*j];
    sequenceVector[OFFSET*j+1] = sequence[OFFSET*j+1];
    // generate Fibonacci numbers.
    for (size_t i = 2; i < OFFSET; ++i)
    {
        size_t elt (OFFSET*j +i);
        sequence[elt] = sequence[elt-1] + sequence[elt - 2];
        sequenceVector[elt] = sequence[elt];
    }
    sequenceVV[j].resize(OFFSET);
    sequenceVV[j] = sequence[std::slice(OFFSET*j,OFFSET,1)];
}

std::valarray<unsigned long> unsignedData(60);
unsigned long base (1 << 31);
std::valarray<double> powers(60);
std::vector<double> powerVector(60);
std::vector<std::valarray<double>> powerVV(3);
std::valarray<std::complex<double>> croots(60);
std::valarray<std::complex<float>> fcroots(60);
std::vector<std::complex<float>> fcroots_vector(60);
std::vector<std::valarray<std::complex<float>>> fcrootv(3);

// create complex data as 60th roots of unity.
double PIBY = M_PI/30.;

for (size_t j = 0; j < rows; ++j)
{
    for (size_t i = 0; i < OFFSET; ++i)
    {
        size_t elt (OFFSET*j+i);
        unsignedData[elt] = sequence[elt];
        croots[elt] = std::complex<double>(std::cos(PIBY*elt),std::sin(PIBY*elt));
        fcroots[elt] = std::complex<float>(croots[elt].real(),croots[elt].imag());
        double x = i+1;
        powers[elt] = pow(x,j+1);
        powerVector[elt] = powers[elt];
    }
    powerVV[j].resize(OFFSET);
    powerVV[j] = powers[std::slice(OFFSET*j,OFFSET,1)];
}

```

```

}

FITSUtil::fill(fcroots_vector,fcroots[std::slice(0,20,1)]);

unsignedData += base;
// syntax identical to Binary Table

Table* newTable = pFits->addTable(hduName,rows,colName,colForm,colUnit);

// numbers is a scalar column

newTable->column(colName[0]).write(numbers,1);

// write valarrays to vector column: note signature change
newTable->column(colName[1]).write(sequence,rows,1);
newTable->column(colName[2]).write(powers,rows,1);
newTable->column(colName[3]).write(unsignedData,rows,1);
newTable->column(colName[4]).write(croots,rows,1);
newTable->column(colName[5]).write(fcroots,rows,3);
newTable->column(colName[6]).write(fcroots_vector,1);
// write vectors to column: note signature change

newTable->column(colName[1]).write(sequenceVector,rows,4);
newTable->column(colName[2]).write(powerVector,rows,4);

std::cout << *newTable << std::endl;

for (size_t j = 0; j < 3 ; ++j)
{
    fcrootv[j].resize(20);
    fcrootv[j] = fcroots[std::slice(20*j,20,1)];
}

// write vector<valarray> object to column.

newTable->column(colName[1]).writeArrays(sequenceVV,7);
newTable->column(colName[2]).writeArrays(powerVV,7);

// create a new vector column in the Table

newTable->addColumn(Tfloat,"powerSeq",20,"none");

// add data entries to it.

newTable->column("powerSeq").writeArrays(powerVV,1);
newTable->column("powerSeq").write(powerVector,rows,4);
newTable->column("dcomplex-roots").write(croots,rows,4);
newTable->column("powerSeq").write(sequenceVector,rows,7);

std::cout << *newTable << std::endl;

```

```
// delete one of the original columns.

newTable->deleteColumn(colName[2]);

// add a new set of rows starting after row 3. So we'll have 14 with
// rows 4,5,6,7,8 blank

newTable->insertRows(3,5);

// now, in the new column, write 3 rows (sequenceVV.size() = 3). This
// will place data in rows 3,4,5 of this column,overwriting them.

newTable->column("powerSeq").writeArrays(sequenceVV,3);
newTable->column("fcomplex-roots").writeArrays(fcrootv,3);

// delete 3 rows starting with row 2. A Table:: method, so the same
// code is called for all Table objects. We should now have 11 rows.

newTable->deleteRows(2,3);

//add a history string. This function call is in HDU:: so is identical
//for all HDUs

    string hist("This file was created for testing CC-
fits write functionality");
    hist += " it serves no other useful purpose. This particu-
lar part of the file was ";
    hist += " constructed to test the writeHistory() and writeCom-
ment() functionality" ;

newTable->writeHistory(hist);

// add a comment string. Use std::string method to change the text in the message
// and write the previous junk as a comment.

hist.insert(0, " COMMENT TEST ");

newTable->writeComment(hist);

// ... print the result.

std::cout << *newTable << std::endl;

return 0;
}
```

2.10 Copying an Extension between Files

Copying extensions from one fits file to another is very straightforward. A complication arises, however, because CCfits requires every FITS object to correspond to a conforming FITS file once constructed. Thus we provide a custom constructor which copies the primary HDU of a “source” FITS file into a new file. Subsequent extensions can be copied by name or extension number as illustrated below.

Note that the simple call

```
FITS::FITS(const std::string& filename)
```

Reads the headers for all of the extensions in the file, so that after the FITS object corresponding to *infile* in the following code is instantiated, all extensions are recognized [read calls are also provided to read only specific HDUs - see below].

In the example code below, the file *outFile* is written straight to disk. Since the code never requests that the HDUs being written to that file are read, the user needs to add statements to do this after the copy is complete.

```
int copyHDU()
{
    //*****
    // copy the 1st and 3rd HDUs from the input file to a new FITS file
    //*****

    const string inFile("atestfil.fit");
    const string outFile("btestfil.fit");

    int status(0);

    status = 0;

    remove(outFile.c_str());           // Delete old file if it already exists

    // open the existing FITS file
    FITS inFile(inFileName);

    // custom constructor FITS::FITS(const string&, const FITS&) for
    // this particular task.

    FITS outFile(outFileName,inFile);

    // copy extension by number...
    outFile.copy(inFile.extension(2));

    // copy extension by name...
    outFile.copy(inFile.extension("TABLE_BINARY"));

    return 0;
}
```

```
}
```

2.11 Selecting Table Data

This function demonstrates the operation of filtering a table by selecting rows that satisfy a condition and writing them to a new file, or overwriting a table with the filtered data. A third mode, where a filtered dataset is appended to the file containing the source data, will be available shortly, but is currently not supported by cfitsio.

The expression syntax for the conditions that may be applied to table data are described in the [cfitsio manual](#). In the example below, we illustrate filtering with a boolean expression involving one of the columns.

The two flags at the end of the call to FITS::filter are an ‘overwrite’ flag - which only has meaning if the inFile and outFile are the same, and a ‘read’ flag. overwrite defaults to true. The second flag is a ‘read’ flag which defaults to false. When set true the user has immediate access to the filtered data.

```
int selectRows()
{
    const string inFile("atestfil.fit");
    const string outFile("btestfil.fit");
    const string newFile("ctestfil.fit");
    remove(newFile.c_str());

    // test 1: write to a new file
    std::auto_ptr<FITS> pInfile(new FITS(inFile,Write,string("PLANETS_ASCII")));
    FITS* infile(pInfile.get());
    std::auto_ptr<FITS> pNewfile(new FITS(newFile,Write));
    ExtHDU& source = infile->extension("PLANETS_ASCII");
    const string expression("DENSITY > 3.0");

    Table& sink1 = pNewfile->filter(expression,source,false,true);

    std::cout << sink1 << std::endl;

    // test 2: write a new HDU to the current file, over-
    // write false, read true.
    // AS OF 7/2/01 does not work because of a bug in cfitsio, but does not
    // crash, simply writes a new header to the file without also writing the
    // selected data.
    Table& sink2 = infile->filter(expression,source,false,true);

    std::cout << sink2 << std::endl;

    // reset the source file back to the extension in question.
```

```
    source = infile->extension("PLANETS_ASCII");

    // test 3: overwrite the current HDU with filtered data.

    Table& sink3 = infile->filter(expression,source,true,true);

    std::cout << sink3 << std::endl;

    return 0;
}
```

2.12 Reading Header information from a HDU

This function demonstrates selecting one HDU from the file, reading the header information and printing out the keys that have been read and the descriptions of the columns.

The `readData` flag is by default false (see below for the alternative case), which means that the data in the column is not read.

```
int readHeader()
{
    const string SPECTRUM("SPECTRUM");

    // read a particular HDU within the file. This call reads just the header
    // information from SPECTRUM

    std::auto_ptr<FITS> pInfile(new FITS("file1.pha",Read,SPECTRUM));

    // define a reference for clarity. (std::auto_ptr<T>::get re-
    turns a pointer

    ExtHDU& table = pInfile->extension(SPECTRUM);

    // read all the keywords, excluding those associ-
    ated with columns.

    table.readAllKeys();

    // print the result.

    std::cout << table << std::endl;

    return 0;
}
```

2.13 Reading an Image

Image reading calls are made very simple: the FITS object is created with the readDataFlag set to true, and reading is done on construction. The following call

```
image.read(contents)
calls
PHDU::read(std::valarray<S>& image).
```

This copies the entire image from the FITS object into the std::valarray object contents, sizing it as necessary. PHDU::read() and ExtHDU::read() [for image extensions] take a range of arguments that support (a) reading the entire image - as in this example; (b) sections of an image starting from a given pixel; (c) rectangular subsets. See the class references for PHDU and ExtHDU for details.

```
int readImage()
{
    std::auto_ptr<FITS> pInfile(new FITS("atestfil.fit",Read,true));

    PHDU& image = pInfile->pHDU();

    std::valarray<unsigned long> contents;

    // read all user-specified, coordinate, and checksum keys in the image
    image.readAllKeys();

    image.read(contents);

    // this doesn't print the data, just header info.
    std::cout << image << std::endl;

    long ax1(image.axis(0));
    long ax2(image.axis(1));

    for (long j = 0; j < ax2; j+=10)
    {
        std::ostream_iterator<short> c(std::cout, "\t");
        std::copy(&contents[j*ax1],&contents[(j+1)*ax1-1],c);
        std::cout << '\n';
    }

    std::cout << std::endl;
    return 0;
}
```

2.14 Reading a Table Extension

Reading table data is similarly straightforward (unsurprisingly, because this application is exactly what CCfits was designed to do easily in the first place).

The two extensions are read on construction, including all the column data [readDataFlag == true] and then printed.

Note that if the data are read as part of the construction, then CCfits uses the row-optimization techniques described in chapter 13 of the cfitsio manual; a chunk of data equal to the size of the available buffer space is read from contiguous disk blocks and transferred to memory storage, as opposed to each column being read in turn. Thus the most efficient way of reading files is to acquire the data on construction.

```

int readTable()
{
    // read a table and explicitly read selected columns. To read instead all the
    // data on construction, set the last argument of the FITS constructor
    // call to 'true'. This functionality was tested in the last release.
    std::vector<string> hdus(2);
    hdus[0] = "PLANETS_ASCII";
    hdus[1] = "TABLE_BINARY";

    std::auto_ptr<FITS> pInfile(new FITS("atestfil.fit",Read,hdus,false));
    ExtHDU& table = pInfile->extension(hdus[1]);

    std::vector < valarray <int > > pp;
    table.column("powerSeq").readArrays( pp, 1,3 );

    std::vector < valarray <std::complex<double> > > cc;
    table.column("dcomplex-roots").readArrays( cc, 1,3 );

    std::valarray < std::complex<float> > ff;
    table.column("fcomplex-roots").read( ff, 4 );

    std::cout << pInfile->extension(hdus[0]) << std::endl;
    std::cout << pInfile->extension(hdus[1]) << std::endl;

    return 0;
}

```

2.15 What's Present, What's Missing

Functionality missing from the initial release falls into two categories: facilities that are designed to be supported but are incompletely implemented, and facilities for which there is no support at present. Functions not supported are, of course, accessible through standard calls to cfitsio. However, calls that modify the data that are not supported by the CCfits library will cause differences between the FITS file and the FITS object. Modifying the fits file using the CCfits interface after using cfitsio calls will likely lead to unexpected results.

In addition, it has not been possible to exercise the entire library before release, particularly because of the many functions implemented as member templates. These are created when an application is linked to the library using either g++ and SunPro CC. Much of the read interface for Table extensions, however, has been tested thoroughly during the development of XSPEC.

Most of the functionality of cfitsio described in Chapter 5 of the cfitsio manual is present, although CCfits is designed to provide atomic read/write operations rather than primitive file manipulation. For example, opening and creating FITS files are private operations which are called by reading and writing constructors. Similarly, errors are treated by C++ exception handling rather than returning status codes, and moving between HDUs within a file is a primitive rather than an atomic operation [in CCfits, operations typically call an internal HDU::makeThisCurrent() call on a specific table or image extension, and then perform the requested read/write operation].

Read/Write operations for keys (in the HDU class) are provided; these implement calls to fits_read_key and fits_update_key respectively. In the case of keywords, which have one of five data types (Integer, Logical, String, Floating and Complex) the user is expected to know the data type of the keyword value in this release. In reading image and table data, intrinsic type conversions are performed as in cfitsio with the exception that reading numeric data into character data is **not** supported. There is an extensive set of member functions supporting equivalents of most of cfitsio's read/write operations: the classes PHDU [primary HDU] and ExtHDU [with subclasses template <typename T> ImageExt<T>], provide multiple overloaded versions of read and write functions. The Column class, instances of which can be held in Table instances [with subclasses AsciiTable and BinTable] has also an extensive set of read/write operations.

A special constructor is provided which creates a new file with the Primary HDU of a source file. A FITS::copy(const HDU&) function copies HDUs from one file into another. Support for filtering table rows by expression is provided by a FITS::filter(...) call which may be used to create a new filtered file or overwrite an existing HDU (see cfitsio manual section 5.6.4).

Functions are provided for adding and deleting columns, and inserting and deleting rows in tables.

HDU objects also have functions to implement writing of history, comment and date keys.

2.15.1 What's Not Present

The coordinate library manipulations [cfitsio manual chapter 7] are not supported.

The iterator work functions [cfitsio manual chapter 6] are not supported. Many of the functions provided are easier to implement using the properties of the standard library, since the standard library containers either allow vectorized operations (in the case of valarrays) or standard library algorithms that take iterators or pointers. In some ways the fits_iterate_data function provide an alternative, approach to the same need

for encapsulation addressed by CCfits.

The hierarchical grouping routines are not supported.

Explicit opening of in-memory data sets as described in Chapter 9 of the manual is *not* supported since none of the FITS constructors call the appropriate cfitsio primitives. The extended file name syntax described in chapters 10 and 11 of the cfitsio manual is not yet supported as it internally creates in-memory objects. This will be supported in a future release.

2.16 Todo List

Class `CCfits::PHDU` Implement functions that allow replacement of the primary image

Member CCfits::FITS::addImage(const string &hduName, int bpix, std::vector< long > &naxes, int version=1)
Add a function for replacing the primary image

Member CCfits::AsciiTable::AsciiTable(FITSBase *p, const string &hduName, int rows, const std::vector< string > &columns)
{enforce equal dimensions for arrays input to AsciiTable, BinTable writing
ctor}

Member ITERATORBASE_DEFECT Oded: Write a main program so that HippoDraw can be installed on the desktop. This requires setting the path to the shared libraries correctly.

Oded: Associate icons with executable and document files. .bmp files for this purpose are in the images subdirectory.

3 CCfits Module Index

3.1 CCfits Modules

Here is a list of all modules:

4 CCfits Hierarchical Index

4.1 CCfits Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

CCfits::FITSUtil::auto_array_ptr	41
CCfits::FITSUtil::CAarray	48
CCfits::Column	51
CCfits::FITSUtil::CVAarray	64
CCfits::FITSUtil::CVarray	64
CCfits::FITS	75
CCfits::FitsException	88
CCfits::Column::InsufficientElements	98
CCfits::Column::InvalidDataType	100
CCfits::Column::InvalidNumberOfRows	103
CCfits::Column::InvalidRowNumber	104
CCfits::Column::InvalidRowParameter	105
CCfits::Column::NoNullValue	112
CCfits::Column::RangeError	124
CCfits::Column::WrongColumnType	131
CCfits::ExtHDU::WrongExtensionType	132
CCfits::FITS::CantCreate	49
CCfits::FITS::CantOpen	50
CCfits::FITS::NoSuchHDU	115
CCfits::FITS::OperationNotSupported	117
CCfits::FitsError	87

CCfits::HDU::InvalidExtensionType	101
CCfits::HDU::InvalidImageDataType	102
CCfits::HDU::NoNullValue	111
CCfits::HDU::NoSuchKeyword	116
CCfits::Table::NoSuchColumn	114
CCfits::FitsFatal	90
CCfits::HDU	91
CCfits::ExtHDU	65
CCfits::Table	125
CCfits::AsciiTable	37
CCfits::BinTable	44
CCfits::PHDU	118
CCfits::Keyword	106
CCfits::FITSUtil::MatchName	109
CCfits::FITSUtil::MatchNum	109
CCfits::FITSUtil::MatchPtrName	110
CCfits::FITSUtil::MatchPtrNum	110
CCfits::FITSUtil::MatchType	111
CCfits::FITSUtil::MatchType::UnrecognizedType	131

5 CCfits Compound Index

5.1 CCfits Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

CCfits::AsciiTable (Class Representing Ascii Table Extensions)	37
---	----

CCfits::FITSUtil::auto_array_ptr (A class that mimics the std:: library auto_ptr class, but works with arrays)	41
CCfits::BinTable (Class Representing Binary Table Extensions. Contains columns with scalar or vector row entries)	44
CCfits::FITSUtil::CAarray (Function object returning C array from a valarray. see CVarray for details)	48
CCfits::FITS::CantCreate (Thrown on failure to create new file)	49
CCfits::FITS::CantOpen (Thrown on failure to open existing file)	50
CCfits::Column (Abstract base class for Column objects)	51
CCfits::FITSUtil::CVAarray (Function object returning C array from a vector of valarrays. see CVarray for details)	64
CCfits::FITSUtil::CVarray (Function object class for returning C arrays from standard library objects used in the FITS library implementation)	64
CCfits::ExtHDU (Base class for all FITS extension HDUs, i.e. Image Extensions and Tables)	65
CCfits::FITS (Memory object representation of a disk FITS file)	75
CCfits::FitsError (FitsError is the exception thrown by non-zero cfitsio status codes)	87
CCfits::FitsException (FitsException is the base class for all exceptions thrown by this library)	88
CCfits::FitsFatal ([potential] base class for exceptions to be thrown on internal library error)	90
CCfits::HDU (Base class for all HDU [Header-Data Unit] objects)	91
CCfits::Column::InsufficientElements (Exception thrown if the data supplied for a write operation is less than declared)	98
CCfits::Column::InvalidDataType (Exception thrown for invalid data type inputs)	100
CCfits::HDU::InvalidExtensionType (Exception to be thrown if user requests extension type that can not be understood as ImageExt , AsciiTable or BinTable)	101

CCfits::HDU::InvalidImageDataType (Exception to be thrown if user requests creation of an image of type not supported by cfitsio)	102
CCfits::Column::InvalidNumberOfRows (Exception thrown if user enters a non-positive number for the number of rows to write)	103
CCfits::Column::InvalidRowNumber (Exception thrown on attempting to read a row number beyond the end of a table)	104
CCfits::Column::InvalidRowParameter (Exception thrown on incorrect row writing request)	105
CCfits::Keyword (Abstract base class implementing the common behavior of Keyword objects)	106
CCfits::FITSUtil::MatchName (Predicate for classes that have a name attribute; match input string with instance name)	109
CCfits::FITSUtil::MatchNum (Predicate for classes that have an index attribute; match input index with instance value)	109
CCfits::FITSUtil::MatchPtrName (As for MatchName , only with the input class a pointer)	110
CCfits::FITSUtil::MatchPtrNum (As for MatchNum , only with the input class a pointer)	110
CCfits::FITSUtil::MatchType (Function object that returns the FITS Value Type corresponding to an input intrinsic type)	111
CCfits::HDU::NoNullValue (Exception to be thrown on seek errors for keywords)	111
CCfits::Column::NoNullValue (Exception thrown if a null value is specified without support from existing column header)	112
CCfits::Table::NoSuchColumn (Exception to be thrown on a failure to retrieve a column specified either by name or index number)	114
CCfits::FITS::NoSuchHDU (Exception thrown by HDU retrieval methods)	115
CCfits::HDU::NoSuchKeyword (Exception to be thrown on seek errors for keywords)	116
CCfits::FITS::OperationNotSupported (Thrown for unsupported operations, such as attempted to select rows from an image extension)	117
CCfits::PHDU (Class representing the primary HDU for a FITS file)	118

CCfits::Column::RangeError (Exception to be thrown for inputs that cause range errors in column read operations)	124
CCfits::Table	125
CCfits::FITSUtil::MatchType::UnrecognizedType (Exception thrown by MatchType if it encounters data type incompatible with cfitsio)	131
CCfits::Column::WrongColumnType (Exception thrown on attempting to access a scalar column as vector data)	131
CCfits::ExtHDU::WrongExtensionType (Exception to be thrown on unmatched extension types)	132

6 CCfits Module Documentation

6.1 FITS Exceptions

Compounds

- class [CCfits::FITS::CantCreate](#)
thrown on failure to create new file.
- class [CCfits::FITS::CantOpen](#)
thrown on failure to open existing file.
- class [CCfits::FitsError](#)
FitsError is the exception thrown by non-zero cfitsio status codes.
- class [CCfits::FitsException](#)
FitsException is the base class for all exceptions thrown by this library.
- class [CCfits::FitsFatal](#)
[potential] base class for exceptions to be thrown on internal library error.
- class [CCfits::HDU::InvalidExtensionType](#)
exception to be thrown if user requests extension type that can not be understood as ImageExt, AsciiTable or BinTable.
- class [CCfits::HDU::InvalidImageDataType](#)
exception to be thrown if user requests creation of an image of type not supported by cfitsio.

- class [CCfits::HDU::NoNullValue](#)
exception to be thrown on seek errors for keywords.
- class [CCfits::Table::NoSuchColumn](#)
Exception to be thrown on a failure to retrieve a column specified either by name or index number.
- class [CCfits::FITS::NoSuchHDU](#)
exception thrown by [HDU](#) retrieval methods.
- class [CCfits::HDU::NoSuchKeyword](#)
exception to be thrown on seek errors for keywords.
- class [CCfits::FITS::OperationNotSupported](#)
thrown for unsupported operations, such as attempted to select rows from an image extension.
- class [CCfits::Column::RangeError](#)
exception to be thrown for inputs that cause range errors in column read operations.
- class [CCfits::FITSUtil::MatchType::UnrecognizedType](#)
exception thrown by [MatchType](#) if it encounters data type incompatible with cfitsio.
- class [CCfits::ExtHDU::WrongExtensionType](#)
Exception to be thrown on unmatched extension types.

7 CCfits Namespace Documentation

7.1 FITSUtil Namespace Reference

FITSUtil is a namespace containing functions used internally by CCfits, but which might be of use for other applications.

7.1.1 Detailed Description

FITSUtil is a namespace containing functions used internally by CCfits, but which might be of use for other applications.

8 CCfits Class Documentation

8.1 CCfits::AsciiTable Class Reference

Class Representing Ascii Table Extensions.

```
#include <AsciiTable.h>
```

Inheritance diagram for CCfits::AsciiTable::

Public Methods

- virtual AsciiTable* **clone** (FITSBase *p) const
 - virtual void **readData** (bool readFlag=false, const std::vector< string > &keys=std::vector< string >())
 - virtual void **addColumn** (ValueType type, const string &columnName, long repeatWidth, const string &colUnit=string("")), long decimals=-1, size_t columnNumber=0)

Protected Methods

- **AsciiTable** (FITSBase *p, const string &hduname=string("”), bool readFlag=false, const std::vector< string > &keys=std::vector< string >(), int version=1)
 - **AsciiTable** (FITSBase *p, const string &hduname, int rows, const std::vector< string > &columnName=std::vector< string >(), const std::vector< string > &columnFmt=std::vector< string >(), const std::vector< string > &columnUnit=std::vector< string >(), int version=1)
 - **AsciiTable** (FITSBase *p, int number)
 - **~AsciiTable** ()

8.1.1 Detailed Description

Class Representing Ascii Table Extensions.

May only contain columns with scalar row entries and a small range of data types. [AsciiTable](#) (re)implements functions prescribed in the [Table](#) abstract class. The implementations allow the calling of cfitsio specialized routines for [AsciiTable](#) header construction.

Direct instantiation of `AsciiTable` objects is disallowed: they are created by explicit calls to `FITS::addTable(...)`, `FITS::read(...)` or internally by one of the `FITS` ctors on initialization. The default for `FITS::addTable` is to produce `BinTable` extensions.

8.1.2 Constructor & Destructor Documentation

8.1.2.1 CCfits::AsciiTable::AsciiTable (FITSBase * *p*, const string & *hduName* = string("")), bool *readFlag* = false, const std::vector< string > & *keys* = std::vector<string>(), int *version* = 1) [protected]

reading constructor: Construct a [AsciiTable](#) extension from an extension of an existing disk file.

The [Table](#) is specified by name and optional version number within the file. An array of strings representing columns or keys indicates which data are to be read. The column data are only read if *readFlag* is true. Reading on construction is optimized, so it is more efficient to read data at the point of instantiation. This favours a "resource acquisition" model of data management.

Parameters:

hduName name of [AsciiTable](#) object to be read.

readFlag flag to determine whether to read data on construction

keys (optional) a list of keywords/columns to be read. The implementation will determine which are keywords. If none are specified, the constructor will simply read the header

version (optional) version number. If not specified, will read the first extension that matches *hduName*.

8.1.2.2 CCfits::AsciiTable::AsciiTable (FITSBase * *p*, const string & *hduName*, int *rows*, const std::vector< string > & *columnName* = std::vector<string>(), const std::vector< string > & *columnFmt* = std::vector<string>(), const std::vector< string > & *columnUnit* = std::vector<string>(), int *version* = 1) [protected]

writing constructor: create new Ascii [Table](#) object with the specified columns.

The constructor creates a valid [HDU](#) which is ready for [Column::write](#) or [insertRows](#) operations. The disk [FITS](#) file is update accordingly. The data type of each column is determined by the *columnFmt* argument (TFORM keywords). See cfitsio documentation for acceptable values.

Parameters:

hduName name of [AsciiTable](#) object to be written

rows number of rows in the table (NAXIS2)

columnName array of column names for columns to be constructed.

columnFmt array of column formats for columns to be constructed.

columnUnit (optional) array of units for data in columns.

version (optional) version number for [HDU](#).

The dimensions of columnType, columnName and columnFmt must match, although this is not enforced at present.

Todo:

{enforce equal dimensions for arrays input to [AsciiTable](#), [BinTable](#) writing ctor}

8.1.2.3 CCfits::AsciiTable::AsciiTable (FITSBase * p, int number)
[protected]

read [AsciiTable](#) with HDU number number from existing file.

This is used internally by methods that need to access HDUs for which no EXTNAME [or equivalent] keyword exists.

8.1.2.4 CCfits::AsciiTable::~AsciiTable () [protected]

destructor.

8.1.3 Member Function Documentation**8.1.3.1 void CCfits::AsciiTable::addColumn (ValueType type, const string & columnName, long repeatWidth, const string & colUnit = string(""), long decimals = -1, size_t columnNumber = 0)** [virtual]

add a new column to an existing table [HDU](#).

Parameters:

type The data type of the column to be added

columnName The name of the column to be added

repeatWidth for a string valued, this is the width of a string. For a numeric column it supplies the vector length of the rows. It is ignored for ascii table numeric data.

colUnit an optional field specifying the units of the data (TUNITn)

decimal optional parameter specifying the number of decimals for an ascii numeric column

columnNumber optional parameter specifying column number to be created. If not specified the column is added to the end. If specified, the column is inserted and the columns already read are reindexed. This parameter is provided as a convenience to support existing code rather than recommended.

Reimplemented from [CCfits::ExtHDU](#).

8.1.3.2 AsciiTable * CCfits::AsciiTable::clone (FITSBase * p) const [virtual]

virtual copy constructor.

Reimplemented from [CCfits::ExtHDU](#).

8.1.3.3 void CCfits::AsciiTable::readData (bool readFlag = false, const std::vector< string > & keys = std::vector<string>()) [virtual]

read columns and keys specified in the input array.

See [Table](#) class documentation for further details.

Reimplemented from [CCfits::ExtHDU](#).

The documentation for this class was generated from the following files:

- [AsciiTable.h](#)
- [AsciiTable.cxx](#)

8.2 CCfits::FITSUtil::auto_array_ptr Class Template Reference

A class that mimics the std:: library auto_ptr class, but works with arrays.

```
#include <FITSUtil.h>
```

Public Methods

- [auto_array_ptr \(X *p=0\) throw \(\)](#)
- [auto_array_ptr \(auto_array_ptr< X > &right\) throw \(\)](#)
- [~auto_array_ptr \(\)](#)
- [void operator= \(auto_array_ptr< X > &right\)](#)
- [X& operator* \(\) throw \(\)](#)
- [X& operator\[\] \(size_t i\) throw \(\)](#)
- [X operator\[\] \(size_t i\) const throw \(\)](#)
- [X* get \(\) const](#)
- [X* release \(\) throw \(\)](#)
- [X* reset \(X *p\) throw \(\)](#)

Static Public Methods

- [void remove \(X *&x\)](#)

8.2.1 Detailed Description

template<typename X> class CCfits::FITSUtil::auto_array_ptr

A class that mimics the std:: library auto_ptr class, but works with arrays.

This code was written by Jack Reeves and first appeared C++ Report, March 1996 edition. Although some authors think one shouldn't need such a contrivance, there seems to be a need for it when wrapping C code.

Usage: replace

float* *f* = new float[200];

with

FITSUtil::auto_array_ptr<float> *f*(new float[200]);

Then the memory will be managed correctly in the presence of exceptions, and delete will be called automatically for *f* when leaving scope.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 template<typename X> CCfits::FITSUtil::auto_array_ptr< X >::auto_array_ptr<X> (X * *p* = 0) throw () [explicit]

constructor. allows creation of pointer to null, can be modified by **reset()**.

8.2.2.2 template<typename X> CCfits::FITSUtil::auto_array_ptr< X >::auto_array_ptr<X> (auto_array_ptr< X > & *right*) throw () [explicit]

copy constructor.

8.2.2.3 template<typename X> CCfits::FITSUtil::auto_array_ptr< X >::~auto_array_ptr<X> ()

destructor.

8.2.3 Member Function Documentation

8.2.3.1 template<typename X> X * CCfits::FITSUtil::auto_array_ptr< X >::get () const

return a token for the underlying content of *this.

8.2.3.2 template<typename X> X & CCfits::FITSUtil::auto_array_ptr< X >::operator * () throw ()

deference operator.

8.2.3.3 template<typename X> void CCfits::FITSUtil::auto_array_ptr< X >::operator= (auto_array_ptr< X > & right)

assignment operator: transfer of ownership semantics.

8.2.3.4] template<typename X> X CCfits::FITSUtil::auto_array_ptr< X >::operator[] (size_t i) const throw ()

return a copy of the ith element of the array.

8.2.3.5] template<typename X> X & CCfits::FITSUtil::auto_array_ptr< X >::operator[] (size_t i) throw ()

return a reference to the ith element of the array.

8.2.3.6 template<typename X> X * CCfits::FITSUtil::auto_array_ptr< X >::release () throw ()

return underlying content of *this, transferring memory ownership.

8.2.3.7 template<typename X> void CCfits::FITSUtil::auto_array_ptr< X >::remove (X *& x) [static]

utility function to delete the memory owned by x and set it to null.

8.2.3.8 template<typename X> X * CCfits::FITSUtil::auto_array_ptr< X >::reset (X * p) throw ()

change the content of the [auto_array_ptr](#) to p.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.3 CCfits::BinTable Class Reference

Class Representing Binary [Table](#) Extensions. Contains columns with scalar or vector row entries.

```
#include <BinTable.h>
```

Inheritance diagram for CCfits::BinTable::

Public Methods

- virtual BinTable* **clone** (FITSBase *p) const
- virtual void **readData** (bool readFlag=false, const std::vector< string > &keys=std::vector< string >())
- virtual void **addColumn** (ValueType type, const string &columnName, long repeatWidth, const string &colUnit=string(""), long decimals=-1, size_t columnNumber=0)

Protected Methods

- **BinTable** (FITSBase *p, const string &hduName=string(""), bool readFlag=false, const std::vector< string > &keys=std::vector< string >(), int version=1)
- **BinTable** (FITSBase *p, const string &hduName, int rows, const std::vector< string > &columnName=std::vector< string >(), const std::vector< string > &columnFmt=std::vector< string >(), const std::vector< string > &columnUnit=std::vector< string >(), int version=1)
- **BinTable** (FITSBase *p, int number)
- **~BinTable** ()

8.3.1 Detailed Description

Class Representing Binary [Table](#) Extensions. Contains columns with scalar or vector row entries.

[BinTable](#) (re)implements functions prescribed in the [Table](#) abstract class. The implementations allow the calling of cfitsio specialized routines for [BinTable](#) header construction. functions particular to the [BinTable](#) class include those dealing with variable width columns

Direct instantiation of [BinTable](#) objects is disallowed: they are created by explicit calls to [FITS::addTable\(... \)](#), [FITS::read\(...\)](#) or internally by one of the [FITS](#) ctors on initialization. For addTable, creation of BinTables is the default.

8.3.2 Constructor & Destructor Documentation

8.3.2.1 CCfits::BinTable::BinTable (FITSBase * p, const string & hduName = string(""), bool readFlag = false, const std::vector< string > & keys = std::vector< string >(), int version = 1) [protected]

reading constructor

Construct a [BinTable](#) extension from an extension of an existing disk file. The [Table](#) is specified by name and optional version number within the file. An array of strings

representing columns or keys indicates which data are to be read. The column data are only read if readFlag is true. Reading on construction is optimized, so it is more efficient to read data at the point of instantiation. This favours a "resource acquisition is initialization" model of data management.

Parameters:

hduName name of [BinTable](#) object to be read.

readFlag flag to determine whether to read data on construction

keys (optional) a list of keywords/columns to be read. The implementation will determine which are keywords. If none are specified, the constructor will simply read the header

version (optional) version number. If not specified, will read the first extension that matches hduName.

8.3.2.2 CCfits::BinTable::BinTable (FITSBase **p*, const string & *hduName*, int *rows*, const std::vector< string > & *columnName* = std::vector<string>(), const std::vector< string > & *columnFmt* = std::vector<string>(), const std::vector< string > & *columnUnit* = std::vector<string>(), int *version* = 1) [protected]

writing constructor.

The constructor creates a valid [HDU](#) which is ready for [Column::write](#) or [insertRows](#) operations. The disk [FITS](#) file is update accordingly. The data type of each column is determined by the columnFmt argument (TFORM keywords). See cfitsio documentation for acceptable values.

Parameters:

hduName name of [BinTable](#) object to be written

rows number of rows in the table (NAXIS2)

columnName array of column names for columns to be constructed.

columnFmt array of column formats for columns to be constructed.

columnUnit (optional) array of units for data in columns.

version (optional) version number for [HDU](#).

The dimensions of columnType, columnName and columnFmt must match, but this is not enforced.

8.3.2.3 CCfits::BinTable::BinTable (FITSBase **p*, int *number*) [protected]

read [BinTable](#) with [HDU](#) number *number* from existing file represented by fitsfile pointer *p*.

8.3.2.4 CCfits::BinTable::~BinTable () [protected]

destructor.

8.3.3 Member Function Documentation**8.3.3.1 void CCfits::BinTable::addColumn (ValueType *type*, const string & *columnName*, long *repeatWidth*, const string & *colUnit* = string("")), long *decimals* = -1, size_t *columnNumber* = 0) [virtual]**

add a new column to an existing table [HDU](#).

Parameters:

type The data type of the column to be added

columnName The name of the column to be added

repeatWidth for a string valued, this is the width of a string. For a numeric column it supplies the vector length of the rows. It is ignored for ascii table numeric data.

colUnit an optional field specifying the units of the data (TUNITn)

decimal optional parameter specifying the number of decimals for an ascii numeric column

columnNumber optional parameter specifying column number to be created. If not specified the column is added to the end. If specified, the column is inserted and the columns already read are reindexed. This parameter is provided as a convenience to support existing code rather than recommended.

Reimplemented from [CCfits::ExtHDU](#).

8.3.3.2 BinTable * CCfits::BinTable::clone (FITSBase * *p*) const [virtual]

virtual copy constructor.

Reimplemented from [CCfits::ExtHDU](#).

8.3.3.3 void CCfits::BinTable::readData (bool *readFlag* = false, const std::vector<string> & *keys* = std::vector<string>()) [virtual]

read columns and keys specified in the input array.

See [Table](#) class documentation for further details.

Reimplemented from [CCfits::ExtHDU](#).

The documentation for this class was generated from the following files:

- BinTable.h
- BinTable.hxx

8.4 CCfits::FITSUtil::CAarray Class Template Reference

function object returning C array from a valarray. see [CVarray](#) for details.

```
#include <FITSUtil.h>
```

Public Methods

- T* [operator\(\)](#) (const std::valarray< T > &inArray)

8.4.1 Detailed Description

template<typename T> class CCfits::FITSUtil::CAarray

function object returning C array from a valarray. see [CVarray](#) for details.

8.4.2 Member Function Documentation

8.4.2.1 template<typename T> T * CCfits::FITSUtil::CAarray< T >::operator() (const std::valarray< T > & inArray)

operator returning C array for use with image data.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.5 CCfits::FITS::CantCreate Class Reference

thrown on failure to create new file.

```
#include <FITS.h>
```

Inheritance diagram for CCfits::FITS::CantCreate::

Public Methods

- [CantCreate](#) (const string &diag, bool silent=false)

8.5.1 Detailed Description

thrown on failure to create new file.

8.5.2 Constructor & Destructor Documentation

8.5.2.1 CCfits::FITS::CantCreate::CantCreate (const string & *diag*, bool *silent* = false)

Exception ctor prefixes the string: "FITS Error: Cannot create file " before specific message.

This exception will be thrown if the user attempts to write to a protected directory or attempts to create a new file with the same name as an existing file without specifying overwrite [overwrite is specified by adding the character '!' before the filename, following the cfitsio convention].

Parameters:

- *msg* A specific diagnostic message, the name of the file that was to be created.
- *silent* if true, print message whether FITS::verboseMode is set or not.

The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

8.6 CCfits::FITS::CantOpen Class Reference

thrown on failure to open existing file.

```
#include <FITS.h>
```

Inheritance diagram for CCfits::FITS::CantOpen::

Public Methods

- [CantOpen](#) (const string &*diag*, bool *silent*=false)

8.6.1 Detailed Description

thrown on failure to open existing file.

8.6.2 Constructor & Destructor Documentation

8.6.2.1 CCfits::FITS::CantOpen::CantOpen (const string & *diag*, bool *silent* = false)

Exception ctor prefixes the string: "FITS Error: Cannot create file " before specific message.

This exception will be thrown if users attempt to open an existing file for write access to which they do not have permission, or of course if the file does not exist.

Parameters:

- *diag* A specific diagnostic message, the name of the file that was to be created.
- *silent* if true, print message whether FITS::verboseMode is set or not.

The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

8.7 CCfits::Column Class Reference

Abstract base class for Column objects.

```
#include <Column.h>
```

Public Methods

- [Column](#) (const Column &right)
- virtual [~Column](#) ()
- virtual void [readData](#) (long firstRow, long nelements, long firstElem=1)=0
- int [rows](#) () const
- void [setDisplay](#) ()
- virtual void [setDimen](#) ()
- int [index](#) () const
- bool [isRead](#) () const
- long [width](#) () const
- size_t [repeat](#) () const
- bool [varLength](#) () const
- double [scale](#) () const
- double [zero](#) () const
- const string& [display](#) () const
- const string& [dimen](#) () const
- ValueType [type](#) () const

- const std::string& **name** () const
- const std::string& **unit** () const
- const std::string& **format** () const
- template<typename S> void **write** (const std::vector< S > &indata, long firstRow)
- template<typename S> void **write** (const std::valarray< S > &indata, long firstRow)
- template<typename S> void **write** (S *indata, long nRows, long firstRow)
- template<typename S> void **write** (const std::vector< S > &indata, long firstRow, S *nullValue)
- template<typename S> void **write** (const std::valarray< S > &indata, long firstRow, S *nullValue)
- template<typename S> void **write** (S *indata, long nRows, long firstRow, S *nullValue)
- template<typename S> void **write** (const std::valarray< S > &indata, long nRows, long firstRow)
- template<typename S> void **write** (const std::vector< S > &indata, long nRows, long firstRow)
- template<typename S> void **write** (S *indata, long nElements, long nRows, long firstRow)
- template<typename S> void **write** (const std::valarray< S > &indata, long nRows, long firstRow, S *nullValue)
- template<typename S> void **write** (const std::vector< S > &indata, long nRows, long firstRow, S *nullValue)
- template<typename S> void **write** (S *indata, long nElements, long nRows, long firstRow, S *nullValue)
- template<typename S> void **write** (const std::valarray< S > &indata, const std::vector< long > &vectorLengths, long firstRow)
- template<typename S> void **write** (const std::vector< S > &indata, const std::vector< long > &vectorLengths, long firstRow)
- template<typename S> void **write** (S *indata, long nElements, const std::vector< long > &vectorLengths, long firstRow)
- template<typename S> void **writeArrays** (const std::vector< std::valarray< S > > &indata, long firstRow)
- template<typename S> void **writeArrays** (const std::vector< std::valarray< S > > &indata, long firstRow, S *nullValue)
- template<typename S> void **read** (std::vector< S > &vals, long first, long last)
- template<typename S> void **read** (std::valarray< S > &vals, long first, long last)
- template<typename S> void **read** (std::valarray< S > &vals, long rows)
- template<typename S> void **readArrays** (std::vector< std::valarray< S > > &vals, long first, long last)
- template<typename S> void **read** (std::vector< S > &vals, long first, long last, S *nullValue)

- template<typename S> void [read](#) (std::valarray< S > &vals, long first, long last, S *nullValue)
- template<typename S> void [read](#) (std::valarray< S > &vals, long rows, S *nullValue)
- template<typename S> void [readArrays](#) (std::vector< std::valarray< S > > &vals, long first, long last, S *nullValue)
- template<typename T> void [addNullValue](#) (T nullVal)

Protected Methods

- [Column](#) (int columnIndex, const string &columnName, ValueType type, const string &format, const string &unit, [Table](#) *p, int rpt=1, long w=1, const string &comment="")
- [Column](#) ([Table](#) *p=0)
- fitsfile* [fitsPointer](#) ()
- void [makeHDUCurrent](#) ()
- virtual std::ostream& [put](#) (std::ostream &s) const
- const std::string& [comment](#) () const

8.7.1 Detailed Description

Abstract base class for [Column](#) objects.

Columns are the data containers used in [FITS](#) tables. Columns of scalar type (one entry per cell) are implemented by the template subclass [ColumnData](#)<T>. Columns of vector type (vector and variable rows) are implemented with the template subclass [ColumnVectorData](#)<T>. [AsciiTables](#) may only contain Columns of type [ColumnData](#)<T>, where T is an implemented [FITS](#) data type (see the [CCfits.h](#) header for a complete list). This requirement is enforced by ensuring that [AsciiTable](#)'s [addColumn](#) method may only create an [AsciiTable](#) compatible column. The [ColumnData](#)<T> class stores its data in a std::vector<T> object.

[BinTables](#) may contain either [ColumnData](#)<T> or [ColumnVectorData](#)<T>. For [ColumnVectorData](#), T must be a numeric type: string vectors are handled by [ColumnData](#)<T>; string arrays are not supported. The internal representation of the data is a std::vector<std::valarray<T>> object. The std::valarray class is designed for efficient numeric processing and has many vectorized numeric and transcendental functions defined on it.

Member template functions for read/write operations are provided in multiple overloads as the interface to data operations. Implicit data type conversions are supported but where they are required make the operations less efficient. Reading numeric column data as character arrays, supported by cfitsio, is not supported by CCfits.

As a base class, [Column](#) provides protected accessor/mutator inline functions to allow only its subclasses to access data members.

8.7.2 Constructor & Destructor Documentation

8.7.2.1 CCfits::Column::Column (const Column & right)

copy constructor, used in copying Columns to standard library containers.

The copy constructor is for internal use only: it does not affect the disk fits file associated with the object.

8.7.2.2 CCfits::Column::~Column () [virtual]

destructor.

8.7.2.3 CCfits::Column::Column (int columnIndex, const string & columnName, ValueType type, const string & format, const string & unit, Table * p, int rpt = 1, long w = 1, const string & comment = "") [protected]

new column creation constructor.

This constructor allows the specification of:

Parameters:

columnIndex The column number

columnName The column name, keyword TTYPEn

type used for determining class of T in ColumnData<T>, ColumnVector-Data<T>

format the column data format, TFORMn keyword

unit the column data unit, TUNITn keyword

p the Table pointer

rpt (optional) repeat count for the row (== 1 for AsciiTables)

w the row width

comment comment to be added to the header.

8.7.2.4 CCfits::Column::Column (Table * p = 0) [protected]

Simple constructor to be called by subclass reading ctors.

8.7.3 Member Function Documentation

8.7.3.1 template<typename T> void CCfits::Column::addNullValue (T nullVal)

Set the TNULLn keyword for the column.

Only relevant for integer valued columns, TNULLn is the value used by cfitsio in undefined processing. All entries in the table equal to an input "null value" are set equal to the value of TNULLn. (For floating point columns a system NaN values is used).

8.7.3.2 const std::string & CCfits::Column::comment () const [inline, protected]

retrieve comment for [Column](#).

8.7.3.3 const string & CCfits::Column::dimen () const [inline]

return TDIMn keyword.

represents dimensions of data arrays in vector columns. for scalar columns, returns a default value.

8.7.3.4 const string & CCfits::Column::display () const [inline]

return TDISPn keyword.

TDISPn is suggested format for output of column data.

8.7.3.5 fitsfile * CCfits::Column::fitsPointer () [protected]

fits pointer corresponding to fits file containing column data.

8.7.3.6 const std::string & CCfits::Column::format () const [inline]

return TFORMn keyword.

TFORMn specifies data format stored in disk file.

8.7.3.7 int CCfits::Column::index () const [inline]

get the [Column](#) index (the n in TTYPEn etc).

8.7.3.8 bool CCfits::Column::isRead () const [inline]

flag set to true if the entire column data has been read from disk.

8.7.3.9 void CCfits::Column::makeHDUCurrent () [protected]

make [HDU](#) containing this the current [HDU](#) of the fits file.

8.7.3.10 const std::string & CCfits::Column::name () const [inline]

return name of [Column](#) (TTYPEEn keyword).

8.7.3.11 std::ostream & CCfits::Column::put (std::ostream & s) const [protected, virtual]

internal implementation of << operator.

8.7.3.12 template<typename S> void CCfits::Column::read (std::valarray< S > & vals, long row, S * nullValue)

return a single row of a vector column into a std::valarray, setting undefined values.

8.7.3.13 template<typename S> void CCfits::Column::read (std::valarray< S > & vals, long first, long last, S * nullValue)

Retrieve data from a scalar column into a std::valarray, setting undefined values.

Parameters:

vals The output container. The function will resize this as necessary

first, last the span of row numbers to read.

8.7.3.14 template<typename S> void CCfits::Column::read (std::vector< S > & vals, long first, long last, S * nullValue)

Retrieve data from a scalar column into a std::vector, setting nullvalue.

As above, only a pointer to a null value is also recognized. If the column is of integer type, then any column value that equals this null value is set equal to the value of the TNULLn keyword. An exception is thrown if TNULLn is not specified. See cfitsio documentation for further details

Parameters:

vals The output container. The function will resize this as necessary

first, last the span of row numbers to read.

nullValue, pointer to integer value regarded as undefined

8.7.3.15 template<typename S> void CCfits::Column::read (std::valarray< S > & vals, long row)

return a single row of a vector column into a std::valarray.

Parameters:

vals The output valarray object

rows The row number to be retrieved (starting at 1).

8.7.3.16 template<typename S> void CCfits::Column::read (std::valarray< S > & *vals*, long *first*, long *last*)

Retrieve data from a scalar column into a std::valarray.

Parameters:

vals The output container. The function will resize this as necessary
first, last the span of row numbers to read.

8.7.3.17 template<typename S> void CCfits::Column::read (std::vector< S > & *vals*, long *first*, long *last*)

Retrieve data from a scalar column into a std::vector.

This and the following functions perform implicit data conversions. An exception will be thrown if no conversion exists.

Parameters:

vals The output container. The function will resize this as necessary
first, last the span of row numbers to read.

8.7.3.18 template<typename S> void CCfits::Column::readArrays (std::vector< std::valarray< S > > & *vals*, long *first*, long *last*, S * *nullValue*)

return a set of rows of a vector column into a container, setting undefined values.

Parameters:

vals The output container. The function will resize this as necessary
first, last the span of row numbers to read.

8.7.3.19 template<typename S> void CCfits::Column::readArrays (std::vector< std::valarray< S > > & *vals*, long *first*, long *last*)

return a set of rows of a vector column into a vector of valarrays.

Parameters:

vals The output container. The function will resize this as necessary
first, last the span of row numbers to read.

8.7.3.20 void CCfits::Column::readData (long *firstrow* = 1, long *nElements* = 1, long *firstelem* = 1) [pure virtual]

read method.

Parameters:

firstRow The first row to be read

firstElem the number of the element on the first row to start at (ignored for scalar columns)

nRows The number of rows to read

8.7.3.21 size_t CCfits::Column::repeat () const [inline]

get the repeat count for the rows.

8.7.3.22 int CCfits::Column::rows () const

return number of rows in the [Column](#).

8.7.3.23 double CCfits::Column::scale () const [inline]

get TSCALn value.

TSCALn is used to convert a data array represented on disk in integer format as floating. Useful for compact storage of digitized data.

8.7.3.24 void CCfits::Column::setDimen () [inline, virtual]

set the TDIMn keyword.

8.7.3.25 void CCfits::Column::setDisplay ()

set the TDISPn keyword.

8.7.3.26 ValueType CCfits::Column::type () const [inline]

returns the data type of the column.

8.7.3.27 const std::string & CCfits::Column::unit () const [inline]

get units of data in [Column](#) (TUNITn keyword).

8.7.3.28 bool CCfits::Column::varLength () const [inline]

boolean, set to true if [Column](#) has variable length vector rows.

8.7.3.29 long CCfits::Column::width () const [inline]

return column data width.

8.7.3.30 template<typename S> void CCfits::Column::write (S * *indata*, long *nElements*, const std::vector<long> & *vectorLengths*, long *firstRow*)

write a C-array of values of size nElements into a column with specified number of entries written per row.

8.7.3.31 template<typename S> void CCfits::Column::write (const std::vector<S> & *indata*, const std::vector<long> & *vectorLengths*, long *firstRow*)

write a vector of values into a column with specified number of entries written per row.

8.7.3.32 template<typename S> void CCfits::Column::write (const std::valarray<S> & *indata*, const std::vector<long> & *vectorLengths*, long *firstRow*)

write a valarray of values into a column with specified number of entries written per row.

Data are written into *vectorLengths.size()* rows, with *vectorLength[n]* elements written to row *n+firstRow -1*. Although clearly designed for wrapping calls to multiple variable-width vector column rows, the code is written to write a variable number of elements to fixed-width column rows (cfitsio requires these operations to be done on a row-by-row basis).

Since cfitsio does not support null value processing for variable width columns this function and its variants do not have version which process undefined values

Parameters:

indata The data to be written

vectorLengths the number of elements to write to each successive row.

firstRow the first row to be written.

8.7.3.33 template<typename S> void CCfits::Column::write (S * *indata*, long *nElements*, long *nRows*, long *firstRow*, S * *nullValue*)

write a C array of values into a range of rows of a vector column, processing undefined values.

8.7.3.34 template<typename S> void CCfits::Column::write (const std::vector<S> & *indata*, long *nRows*, long *firstRow*, S * *nullValue*)

write a vector of values into a range of rows of a vector column, processing undefined values.

see valarray version for details.

8.7.3.35 template<typename S> void CCfits::Column::write (const std::valarray< S > & *indata*, long *nRows*, long *firstRow*, S * *nullValue*)

write a valarray of values into a range of rows of a vector column.

see version without undefined processing for details.

8.7.3.36 template<typename S> void CCfits::Column::write (S * *indata*, long *nElements*, long *nRows*, long *firstRow*)

write a C array of values into a range of rows of a vector column.

Details are as for vector input; only difference is the need to supply the size of the C-array.

Parameters:

indata The data to be written.

nElements The size of *indata*

nRows the number of rows to which to write the data.

firstRow The first row to be written

8.7.3.37 template<typename S> void CCfits::Column::write (const std::vector< S > & *indata*, long *nRows*, long *firstRow*)

write a vector of values into a range of rows of a vector column.

see valarray version for details.

8.7.3.38 template<typename S> void CCfits::Column::write (const std::valarray< S > & *indata*, long *nRows*, long *firstRow*)

write a valarray of values into a range of rows of a vector column.

This and the equivalent vector version write a vector of values into *nRows*. The primary use of this is for fixed width columns, in which case [Column](#)'s repeat attribute is used to determine how many elements are written to each row; if *indata.size()* is too small an exception will be thrown. If the column is variable width, the call will write *indata.size()/nRows* elements to each row.

Parameters:

indata The data to be written.

nRows the number of rows to which to write the data.

firstRow The first row to be written

8.7.3.39 template<typename S> void CCfits::Column::write (S * *inData*, long *nRows*, long *firstRow*, S * *nullValue*)

write a C array into a scalar [Column](#), processing undefined values.

Parameters:

inData The data to be written.

nRows The size of the data array to be written

firstRow The first row to be written

nullValue Pointer to the value in the input array to be set to undefined values

8.7.3.40 template<typename S> void CCfits::Column::write (const std::valarray< S > & *inData*, long *firstRow*, S * *nullValue*)

write a valarray of values into a scalar column starting with *firstRow* with undefined values set to *nullValue*.

Parameters:

inData The data to be written.

firstRow The first row to be written

nullValue Pointer to the value in the input array to be set to undefined values

8.7.3.41 template<typename S> void CCfits::Column::write (const std::vector< S > & *inData*, long *firstRow*, S * *nullValue*)

write a vector of values into a row starting with *firstRow* with undefined values set to *nullValue*.

Parameters:

inData The data to be written.

firstRow The first row to be written

nullValue Pointer to the value in the input array to be set to undefined values

8.7.3.42 template<typename S> void CCfits::Column::write (S * *indata*, long *nRows*, long *firstRow*)

write a C array of size *nRows* into a scalar [Column](#) starting with row *firstRow*.

Parameters:

indata The data to be written.

nRows The size of the data array to be written

firstRow The first row to be written

8.7.3.43 template<typename S> void CCfits::Column::write (const std::valarray< S > & *indata*, long *firstRow*)

write a valarray of values into a scalar column starting with *firstRow*.

Parameters:

indata The data to be written.

firstRow The first row to be written

8.7.3.44 template<typename S> void CCfits::Column::write (const std::vector< S > & *indata*, long *firstRow*)

write a vector of values into a row starting with *firstRow*.

Parameters:

indata The data to be written.

firstRow The first row to be written

8.7.3.45 template<typename S> void CCfits::Column::writeArrays (const std::vector< std::valarray< S > > & *indata*, long *firstRow*, S * *nullValue*)

write a vector of valarray objects to the column, starting at row *firstRow* ≥ 1 , processing undefined values.

8.7.3.46 template<typename S> void CCfits::Column::writeArrays (const std::vector< std::valarray< S > > & *indata*, long *firstRow*)

write a vector of valarray objects to the column, starting at row *firstRow* ≥ 1 .

8.7.3.47 double CCfits::Column::zero () const [inline]

get TZEROOn value.

TZEROOn is an integer offset used in the implementation of unsigned data

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx
- ColumnT.h

8.8 CCfits::FITSUtil::CVAarray Class Template Reference

function object returning C array from a vector of valarrays. see [CVarray](#) for details.

```
#include <FITSUtil.h>
```

Public Methods

- `T* operator() (const std::vector< std::valarray< T > > &inArray)`

8.8.1 Detailed Description

template<typename T> class CCfits::FITSUtil::CVAarray

function object returning C array from a vector of valarrays. see [CVarray](#) for details.

8.8.2 Member Function Documentation

8.8.2.1 template<typename T> T * CCfits::FITSUtil::CVAarray< T >::operator() (const std::vector< std::valarray< T > > & inArray)

operator returning C array for use with vector column data.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.9 CCfits::FITSUtil::CVarray Class Template Reference

Function object class for returning C arrays from standard library objects used in the [FITS](#) library implementation.

```
#include <FITSUtil.h>
```

Public Methods

- `T* operator() (const std::vector< T > &inArray)`

8.9.1 Detailed Description

template<typename T> class CCfits::FITSUtil::CVarray

Function object class for returning C arrays from standard library objects used in the **FITS** library implementation.

There are 3 versions which convert `std::vector<T>`, `std::valarray<T>`, and `std::vector<std::valarray<T>>` objects to pointers to `T`, called **CVarray**, **CAarray**, and **CVAarray**.

An alternative function, `CharArray`, is provided to deal with the special case of vector string arrays.

8.9.2 Member Function Documentation

8.9.2.1 template<typename T> T * CCfits::FITSUtil::CVarray< T >::operator() (const std::vector< T > & inArray) [inline]

operator returning C array for use with scalar column data.

The documentation for this class was generated from the following file:

- `FITSUtil.h`

8.10 CCfits::ExtHDU Class Reference

base class for all **FITS** extension HDUs, i.e. Image Extensions and Tables.

```
#include <ExtHDU.h>
```

Inheritance diagram for CCfits::ExtHDU::

Public Methods

- `ExtHDU (const ExtHDU &right)`
- virtual `~ExtHDU ()`
- virtual void `readData (bool readFlag=false, const std::vector< string > &keys=std::vector< string >())=0`

- const string& **name** () const
- virtual **HDU*** **clone** (FITSBase *p) const=0
- virtual void **makeThisCurrent** () const
- virtual **Column& column** (const string &colName) const
- virtual **Column& column** (int colIndex) const
- virtual long **rows** () const
- virtual void **addColumn** (ValueType type, const string &columnName, long repeatWidth, const string &colUnit=string(""), long decimals=-1, size_t columnNumber=0)
- virtual void **deleteColumn** (const string &columnName)
- int **version** () const
- void **version** (int value)
- std::string& **name** ()
- template<typename S> void **write** (const std::vector< long > &first, long nElements, const std::valarray< S > &data, S *nullValue)
- template<typename S> void **write** (long first, long nElements, const std::valarray< S > &data, S *nullValue)
- template<typename S> void **write** (const std::vector< long > &first, long nElements, const std::valarray< S > &data)
- template<typename S> void **write** (long first, long nElements, const std::valarray< S > &data)
- template<typename S> void **write** (const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::valarray< S > &data)
- template<typename S> void **read** (std::valarray< S > &image)
- template<typename S> void **read** (std::valarray< S > &image, long first, long nElements, S *nullValue)
- template<typename S> void **read** (std::valarray< S > &image, const std::vector< long > &first, long nElements, S *nullValue)
- template<typename S> void **read** (std::valarray< S > &image, const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::vector< long > &stride)
- template<typename S> void **read** (std::valarray< S > &image, long first, long nElements)
- template<typename S> void **read** (std::valarray< S > &image, const std::vector< long > &first, long nElements)
- template<typename S> void **read** (std::valarray< S > &image, const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::vector< long > &stride, S *nullValue)

Static Public Methods

- void **readHduName** (const fitsfile *fptr, int hduIndex, string &hduName, int &hduVersion)

Protected Methods

- `ExtHDU` (`FITSBase *p=0, HduType xtype=AnyHdu, const string &hduName=string("")`, int `version=1`)
- `ExtHDU` (`FITSBase *p, HduType xtype, const string &hduName, int bitpix, int naxis, const std::vector< long > &axes, int version=1`)
- `ExtHDU` (`FITSBase *p, HduType xtype, int number`)
- long `pcount` () const
- void `pcount` (long value)
- long `gcount` () const
- void `gcount` (long value)
- HduType `xtension` () const
- void `xtension` (HduType value)

8.10.1 Detailed Description

base class for all `FITS` extension HDUs, i.e. Image Extensions and Tables.

`ExtHDU` needs to have the combined public interface of `Table` objects and images. It achieves this by providing the same set of read and write operations as `PHDU`, and also providing the same operations for extracting columns from the extension as does `Table` [after which the column interface is accessible]. Differentiation between extension types operates by exception handling: .i.e. attempting to access image data structures on a `Table` object through the `ExtHDU` interface will or trying to return a `Column` reference from an Image extension will both throw an exception

8.10.2 Constructor & Destructor Documentation

8.10.2.1 CCfits::ExtHDU::ExtHDU (const ExtHDU & right)

copy constructor.

8.10.2.2 CCfits::ExtHDU::~ExtHDU () [virtual]

destructor.

8.10.2.3 CCfits::ExtHDU::ExtHDU (FITSBase * p = 0, HduType xtype = AnyHdu, const string & hduName = string("")), int version = 1) [protected]

default constructor, required as Standard Library Container content.

8.10.2.4 CCfits::ExtHDU::ExtHDU (FITSBase * p, HduType xtype, const string & hduName, int bitpix, int naxis, const std::vector< long > & axes, int version = 1) [protected]

writing constructor.

The writing constructor forces the user to supply a name for the [HDU](#). The bitpix, naxes and naxis data required by this constructor are required [FITS](#) keywords for any HDUs.

8.10.2.5 CCfits::ExtHDU::ExtHDU ([FITSBase * p](#), [HduType xtype](#), [int number](#)) [protected]

[ExtHDU](#) constructor for getting ExtHDUs by number.

Necessary since EXTNAME is a reserved, not required, keyword. But a synthetic name is supplied by static [ExtHDU::readHduName](#) which is called by this constructor.

8.10.3 Member Function Documentation

8.10.3.1 void CCfits::ExtHDU::addColumn ([ValueType type](#), [const string & columnName](#), [long repeatWidth](#), [const string & colUnit = string\(""\)](#), [long decimals = -1](#), [size_t columnNumber = 0](#)) [virtual]

add a new column to an existing table [HDU](#).

Parameters:

type The data type of the column to be added

columnName The name of the column to be added

repeatWidth for a string valued, this is the width of a string. For a numeric column it supplies the vector length of the rows. It is ignored for ascii table numeric data.

colUnit an optional field specifying the units of the data (TUNITn)

decimal optional parameter specifying the number of decimals for an ascii numeric column

columnNumber optional parameter specifying column number to be created. If not specified the column is added to the end. If specified, the column is inserted and the columns already read are reindexed. This parameter is provided as a convenience to support existing code rather than recommended.

Reimplemented in [CCfits::AsciiTable](#), and [CCfits::BinTable](#).

8.10.3.2 [HDU * CCfits::ExtHDU::clone](#) ([FITSBase * p](#)) const [pure virtual]

virtual copy constructor.

Reimplemented from [CCfits::HDU](#).

Reimplemented in [CCfits::AsciiTable](#), and [CCfits::BinTable](#).

8.10.3.3 Column & CCfits::ExtHDU::column (int colIndex) const [virtual]

return a reference to a [Table](#) column specified by column index.

This version is provided for convenience; the 'return by name' version is more efficient because columns are stored in an associative array sorted by name.

Exceptions:

WrongExtensionType thrown if *this is an image extension.

Reimplemented in [CCfits::Table](#).

8.10.3.4 Column & CCfits::ExtHDU::column (const string & colName) const [virtual]

return a reference to a [Table](#) column specified by name.

The overridden base class implementation [ExtHDU::column](#) throws an exception, which is thus the action to be taken if self is an image extension

Exceptions:

WrongExtensionType see above

Reimplemented in [CCfits::Table](#).

8.10.3.5 void CCfits::ExtHDU::deleteColumn (const string & name) [virtual]

delete a column in a [Table](#) extension by name.

Parameters:

columnName The name of the column to be deleted.

Exceptions:

WrongExtensionType if extension is an image.

Reimplemented in [CCfits::Table](#).

8.10.3.6 void CCfits::ExtHDU::gcount (long value) [inline, protected]

set required gcount keyword value.

8.10.3.7 long CCfits::ExtHDU::gcount () const [inline, protected]

return required gcount keyword value.

8.10.3.8 void makeThisCurrent () const [virtual]

move the fitsfile pointer to this current [HDU](#).

This function should never need to be called by the user since it is called internally whenever required.

Reimplemented from [CCfits::HDU](#).

8.10.3.9 std::string & CCfits::ExtHDU::name () [inline]

set/return the extension name.

8.10.3.10 const string & CCfits::ExtHDU::name () const [inline]

return the name of the extension.

8.10.3.11 void CCfits::ExtHDU::pcount (long value) [inline, protected]

set required pcount keyword value.

8.10.3.12 long CCfits::ExtHDU::pcount () const [inline, protected]

return required pcount keyword value.

8.10.3.13 template<typename S> void CCfits::ExtHDU::read (std::valarray< S > & image, const std::vector< long > & firstVertex, const std::vector< long > & lastVertex, const std::vector< long > & stride, S * nulValue)

read an image subset into valarray image, processing null values.

The image subset is defined by two vertices and a stride indicating the 'densemess' of the values to be picked in each dimension (a stride = (1,1,1,...) means picking every pixel in every dimension, whereas stride = (2,2,2,...) means picking every other value in each dimension.

8.10.3.14 template<typename S> void CCfits::ExtHDU::read (std::valarray< S > & image, const std::vector< long > & first, long nElements)

read an image section starting at a location specified by an n-tuple.

8.10.3.15 template<typename S> void CCfits::ExtHDU::read (std::valarray< S > & image, long first, long nElements)

read an image section starting at a specified pixel.

8.10.3.16 template<typename S> void CCfits::ExtHDU::read (std::valarray< S > & *image*, const std::vector< long > & *firstVertex*, const std::vector< long > & *lastVertex*, const std::vector< long > & *stride*)

read an image subset.

8.10.3.17 template<typename S> void CCfits::ExtHDU::read (std::valarray< S > & *image*, const std::vector< long > & *first*, long *nElements*, S * *nullValue*)

read part of an image array, processing null values.

As above except for

Parameters:

first a vector<long> representing an n-tuple giving the coordinates in the image of the first pixel.

8.10.3.18 template<typename S> void CCfits::ExtHDU::read (std::valarray< S > & *image*, long *first*, long *nElements*, S * *nullValue*)

read part of an image array, processing null values.

Implicit data conversion is supported (i.e. user does not need to know the type of the data stored. A [WrongExtensionType](#) exception is thrown if *this is not an image.

Parameters:

image The receiving container, a std::valarray reference

first The first pixel from the array to read [a long value]

nElements The number of values to read

nullValue A pointer containing the value in the table to be considered as undefined. See cfitsio for details

8.10.3.19 template<typename S> void CCfits::ExtHDU::read (std::valarray< S > & *image*)

Read image data into container.

The container image contains the entire image array after the call. This and all the other variants of `read()` throw a [WrongExtensionType](#) exception if called for a [Table](#) object.

8.10.3.20 void CCfits::ExtHDU::readData (bool *readFlag* = false, const std::vector< string > & *keys* = std::vector<string>()) [pure virtual]

read data from [HDU](#) depending on *readFlag* and *keys*.

Reimplemented in [CCfits::AsciiTable](#), and [CCfits::BinTable](#).

8.10.3.21 void CCfits::ExtHDU::readHduName (const fitsfile * *fptr*, int *hduIndex*, string & *hduName*, int & *hduVersion*) [static]

read extension name.

Used primarily to allow extensions to be specified by **HDU** number and provide their name for the associative array that contains them. Alternatively, if there is no name keyword in the extension, one is synthesized from the index.

8.10.3.22 long CCfits::ExtHDU::rows () const [virtual]

return the number of rows in the extension.

Exceptions:

WrongExtensionType thrown if *this is an image extension.

Reimplemented in [CCfits::Table](#).

8.10.3.23 void CCfits::ExtHDU::version (int *value*) [inline]

set the extension version number.

8.10.3.24 int CCfits::ExtHDU::version () const [inline]

return the extension version number.

8.10.3.25 template<typename S> void CCfits::ExtHDU::write (const std::vector< long > & *firstVertex*, const std::vector< long > & *lastVertex*, const std::valarray< S > & *data*)

write a subset (generalize slice) of data to the image.

A generalized slice/subset is a subset of the image (e.g. one plane of a data cube of size <= the dimension of the cube). It is specified by two opposite vertices. The equivalent cfitsio call does not support undefined data processing so there is no version that allows a null value to be specified.

Parameters:

firstVertex the coordinates specifying lower and upper vertices of the n-dimensional slice

lastVertex

data The data to be written

8.10.3.26 template<typename S> void CCfits::ExtHDU::write (long *first*, long *nElements*, const std::valarray< S > & *data*)

write array starting from specified pixel number, without undefined data processing.

8.10.3.27 template<typename S> void CCfits::ExtHDU::write (const std::vector< long > & *first*, long *nElements*, const std::valarray< S > & *data*)

write array starting from specified n-tuple, without undefined data processing.

8.10.3.28 template<typename S> void CCfits::ExtHDU::write (long *first*, long *nElements*, const std::valarray< S > & *data*, S * *nullValue*)

write array to image starting with a specified pixel and allowing undefined data to be processed.

parameters after the first are as for version with n-tuple specifying first element. these two version are equivalent, except that it is possible for the first pixel number to exceed the range of 32-bit integers, which is how long datatype is commonly implemented.

8.10.3.29 template<typename S> void CCfits::ExtHDU::write (const std::vector< long > & *first*, long *nElements*, const std::valarray< S > & *data*, S * *nullValue*)

Write a set of pixels to an image extension with the first pixel specified by an n-tuple, processing undefined data.

All the overloaded versions of `ExtHDU::write` perform operations on `*this` if it is an image and throw a `WrongExtensionType` exception if not. Where appropriate, alternate versions allow undefined data to be processed

Parameters:

first an n-tuple of dimension equal to the image dimension specifying the first pixel in the range to be written

nElements number of pixels to be written

data array of data to be written

pointer to null value (data with this value written as undefined; needs the BLANK keyword to have been specified).

8.10.3.30 void CCfits::ExtHDU::xtension (HduType *value*) [inline, protected]

set the extension type.

8.10.3.31 HduType CCfits::ExtHDU::xtension () const [inline, protected]

return the extension type.

allowed values are ImageHDU, AsciiTbl, and BinaryTbl

The documentation for this class was generated from the following files:

- ExtHDU.h
- ExtHDU.cxx
- ExtHDUT.h
- FITS.h

8.11 CCfits::FITS Class Reference

Memory object representation of a disk **FITS** file.

```
#include <FITS.h>
```

Public Methods

- **FITS** (const string &name, RWmode mode=Read, bool readDataFlag=false, const std::vector< string > &primaryKeys=std::vector< string >())
- **FITS** (const string &name, RWmode mode, const string &hduName, bool readDataFlag=false, const std::vector< string > &hduKeys=std::vector< string >(), const std::vector< string > &primaryKey=std::vector< string >(), int version=1)
- **FITS** (const string &name, RWmode mode, const std::vector< string > &hduNames, bool readDataFlag=false, const std::vector< string > &primaryKey=std::vector< string >())
- **FITS** (const string &fileName, const FITS &source)
- **FITS** (const string &name, RWmode mode, const std::vector< string > &hduNames, const std::vector< std::vector< string > > &hduKeys, bool readDataFlag=false, const std::vector< string > &primaryKeys=std::vector< string >(), const std::vector< int > &hduVersions=std::vector< int >())
- **FITS** (const string &name, int bitpix, int naxis, long *naxes)
- **FITS** (const string &name, RWmode mode, int hduIndex, bool readDataFlag=false, const std::vector< string > &hduKeys=std::vector< string >(), const std::vector< string > &primaryKey=std::vector< string >())
- **FITS** (const string &name, RWmode mode, const std::vector< string > &searchKeys, const std::vector< string > &searchValues, bool readDataFlag=false, const std::vector< string > &hduKeys=std::vector< string >(), const std::vector< string > &primaryKey=std::vector< string >(), int version=1)
- **~FITS** ()
- void **deleteExtension** (const std::string &doomed, int version=1)
- void **read** (const string &hduName, bool readDataFlag=false, const std::vector< string > &keys=std::vector< string >(), int version=0)
- void **read** (const std::vector< string > &hduNames, bool readDataFlag=false)

- void `read` (const std::vector< string > &hduNames, const std::vector< std::vector< string > > &keys, bool readDataFlag=false, const std::vector< int > &hduVersions=std::vector< int >())
- FITS* `clone` () const
- const ExtHDU& `extension` (int i) const
- ExtHDU& `extension` (int i)
- const ExtHDU& `extension` (const string &hduName, int version=1) const
- const PHDU& `pHDU` () const
- PHDU& `pHDU` ()
- ExtHDU& `extension` (const string &hduName, int version=1)
- void `read` (int hduIndex, bool readDataFlag=false, const std::vector< string > &keys=std::vector< string >())
- Table* `addTable` (string &hduName, int rows, const std::vector< string > &columnName=std::vector< string >(), const std::vector< string > &columnFmt=std::vector< string >(), const std::vector< string > &columnUnit=std::vector< string >(), HduType type=BinaryTbl, int version=1)
- ExtHDU* `addImage` (const string &hduName, int bpix, std::vector< long > &naxes, int version=1)
- void `destroy` () throw ()
- void `flush` ()
- const string& `currentExtensionName` () const
- void `read` (const std::vector< string > &searchKeys, const std::vector< string > &searchValues, bool readDataFlag=false, const std::vector< string > &hduKeys=std::vector< string >(), int version=1)
- const std::multimap<string,ExtHDU*>& `extension` () const
- void `resetPosition` ()
- const string& `name` () const
- void `copy` (const HDU &source)
- Table& `filter` (const string &expression, ExtHDU &inputTable, bool overwrite=true, bool readData=false)
- ExtHDU& `currentExtension` ()
- void `deleteExtension` (int doomed)

Static Public Methods

- void `clearErrors` ()
- bool `verboseMode` ()
- void `setVerboseMode` (bool value)

8.11.1 Detailed Description

Memory object representation of a disk **FITS** file.

Constructors are provided to get **FITS** data from an existing file or to create new **FITS** data sets. Overloaded versions allow the user to

- a) read from one or more specified extensions, specified by EXTNAME and VERSION or by **HDU** number.
- b) either just header information or data on construction
- c) to specify scalar keyword values to be read on construction
- d) to open and read an extension that has specified keyword values
- e) create a new **FITS** object and corresponding file, including an empty primary header.

The memory fits object as constructed is always an image of a valid **FITS** object, i.e. a primary **HDU** is created on construction.

calling the destructor closes the disk file, so that **FITS** files are automatically deleted at the end of scope unless other arrangements are made.

8.11.2 Constructor & Destructor Documentation

8.11.2.1 CCfits::FITS::FITS (const string & name, RWmode mode = Read, bool readDataFlag = false, const std::vector< string > & primaryKeys = std::vector<string>())

basic constructor.

This basic constructor makes a **FITS** object from the given filename. The filename string is passed directly to the cfitsio library: thus all of the extended filename syntax described in the cfitsio manual should work as documented.

If the mode is Read [default], it will read all of the headers in the file, and all of the data if the readDataFlag is supplied as true. It will also read optional primary keys.

The file name is the only required argument. If the mode is Write and the file does not already exist, a default primary **HDU** will be created in the file with BITPIX=8 and NAXIS=0: this mode is designed for writing **FITS** files with table extensions only. For files with image data the constructor that specified the data type and number of axes should be called.

Parameters:

name The name of the **FITS** file to be read/written

mode The read/write mode: must be Read or Write

readDataFlag boolean: read data on construction if true

primaryKeys Allows optional reading of primary header keys on construction

Exceptions:

NoSuchHDU thrown on **HDU** seek error either by index or {name,version}

FitsError thrown on non-zero status code from cfitsio when not overriden by **FitsException** error to produce more illuminating message.

8.11.2.2 CCfits::FITS (const string & *name*, RWmode *mode*, const string & *hduName*, bool *readDataFlag* = false, const std::vector< string > & *hduKeys* = std::vector<string>(), const std::vector< string > & *primaryKey* = std::vector<string>(), int *version* = 1)

Open a **FITS** file and read a single specified **HDU**.

This and similar constructor variants support reading table data.

Optional arguments allow the reading of primary header keys and specified data from *hduName*, the **HDU** to be read. An object representing the primary **HDU** is always created: if it contains an image, that image may be read by subsequent calls.

Parameters:

name The name of the **FITS** file to be read

mode The read/write mode: takes values Read or Write

hduName The name of the **HDU** to be read.

hduKeys Optional array of keywords to be read from the **HDU**

version Optional version number. If not supplied the first **HDU** with name *hduName* is read see above for other parameter definitions

8.11.2.3 CCfits::FITS (const string & *name*, RWmode *mode*, const std::vector< string > & *hduNames*, bool *readDataFlag* = false, const std::vector< string > & *primaryKey* = std::vector<string>())

This is intended as a convenience where the file consists of single versions of HDUs and data only, not keys are to be read.

Parameters:

hduNames array of **HDU** names to be read. see above for other parameter definitions.

8.11.2.4 CCfits::FITS (const string & *fileName*, const **FITS** & *source*)

create a new **FITS** object and corresponding file with copy of the primary header of the source.

Parameters:

fileName New file to be created.

source A previously created **FITS** object to be copied.

see above for other parameter definitions.

8.11.2.5 CCfits::FITS (const string & *name*, RWmode *mode*, const std::vector< string > & *hduNames*, const std::vector< std::vector< string > > & *hduKeys*, bool *readDataFlag* = false, const std::vector< string > & *primaryKeys* = std::vector<string>(), const std::vector< int > & *hduVersions* = std::vector<int>())

FITS read constructor in full generality.

Parameters:

hduVersions an optional version number for each HDU to be read

hduKeys an array of keywords for each HDU to be read. see above for other parameter definitions.

8.11.2.6 CCfits::FITS (const string & *name*, int *bitpix*, int *naxis*, long * *naxes*)

Constructor for creating new FITS objects containing images.

This constructor is only called for creating new files (mode is not an argument) and creates a new primary HDU with the datatype & axes specified by bitpix, naxis, and naxes. The data are added to the new fits object and file by subsequent calls to **FITS::p-HDU().write(<arguments>)**

Valid values of bitpix are given in the fitsio.h header. If the filename corresponds to an existing file and does not start with the '!' character the construction will fail with a **CantCreate** exception.

The arguments are:

Parameters:

name The file to be written to disk

bitpix the datatype of the primary image (see cfitsio documentation for allowed values.

naxis the data dimension of the primary image

naxes the array of axis lengths for the primary image. Ignored if naxis =0, i.e. the primary header is empty. extensions can be added arbitrarily to the file after this constructor is called. The constructors should write header information to disk:

8.11.2.7 CCfits::FITS (const string & *name*, RWmode *mode*, int *hduIndex*, bool *readDataFlag* = false, const std::vector< string > & *hduKeys* = std::vector<string>(), const std::vector< string > & *primaryKey* = std::vector<string>())

read a single numbered HDU.

Constructor analogous to the version that reads by name. This is required since **HDU** extensions are not required to have the EXTNAME or HDUNAME keyword by the standard. If there is no name, a dummy name based on the **HDU** number is created and becomes the key.

Parameters:

hduIndex The index of the **HDU** to be read. see above for other parameter definitions.

8.11.2.8 CCfits::FITS (const string & name, RWmode mode, const std::vector< string > & searchKeys, const std::vector< string > & searchValues, bool readDataFlag = false, const std::vector< string > & hduKeys = std::vector<string>(), const std::vector< string > & primaryKey = std::vector<string>(), int version = 1)

open fits file and read **HDU** which contains supplied keywords with [optional] specified values (sometimes one just wants to know that the keyword is present).

Optional parameters allows the reading of specified primary **HDU** keys and specified columns and keywords in the **HDU** of interest.

Parameters:

name The name of the **FITS** file to be read

mode The read/write mode: must be Read or Write

searchKeys A string vector of keywords to search for in each header

searchValues A string vector of values those keywords are required to have for success. Note that the keys must be of type string. If any value does not need to be checked the corresponding searchValue element can be empty.

readDataFlag boolean: if true, read data if **HDU** is found

hduKeys Allows optional reading of keys in the **HDU** that is searched for if it is successfully found

primaryKeys Allows optional reading of primary header keys on construction

version Optional version number. If specified, checks the EXTVERS keyword.

Exceptions:

FitsError thrown on non-zero status code from cfitsio when not overridden by **FitsException** error to produce more illuminating message.

8.11.2.9 CCfits::FITS::~FITS ()

destructor.

8.11.3 Member Function Documentation

8.11.3.1 ExtHDU * CCfits::FITS::addImage (const string & *hduName*, int *bpix*, std::vector< long > & *naxes*, int *version* = 1)

Add an image extension to an existing [FITS](#) object. (File with w or rw access).

Does not make primary images, which are built in the constructor for the [FITS](#) file. The image data is not added here: it can be added by a second call.

Todo:

Add a function for replacing the primary image

8.11.3.2 Table * CCfits::FITS::addTable (string & *hduName*, int *rows*, const std::vector< string > & *columnName* = std::vector<string>(), const std::vector< string > & *columnFmt* = std::vector<string>(), const std::vector< string > & *columnUnit* = std::vector<string>(), HduType *type* = BinaryTbl, int *version* = 1)

Add a table extension to an existing [FITS](#) object. Add extension to [FITS](#) object for file with w or rw access.

Parameters:

rows The number of rows in the table to be created.

columnName A vector containing the table column names

columnFmt A vector containing the table column formats

columnUnit (Optional) a vector giving the units of the columns.

HduType The table type - AsciiTbl or BinaryTbl (defaults to BinaryTbl) the lists of columns are optional - one can create an empty table extension but if supplied, colType, columnName and colFmt must have equal dimensions.

Todo:

the code should one day check that the version keyword is higher than any other versions already added to the [FITS](#) object (although cfitsio doesn't do this either).

8.11.3.3 void CCfits::FITS::clearErrors () [static]

clear the error stack and set status to zero.

8.11.3.4 FITS * CCfits::FITS::clone () const

clone function: to allow implementation later of reference counting.

8.11.3.5 void CCfits::FITS::copy (const HDU & source)

copy the **HDU** source into the **FITS** object.

This function adds a copy of an **HDU** from another file into *this. It does not create a duplicate of an **HDU** in the file associated with *this.

8.11.3.6 ExtHDU & CCfits::FITS::currentExtension ()

return a non-const reference to whichever is the current extension.

8.11.3.7 const string & CCfits::FITS::currentExtensionName () const

return the name of the extension that the fitsfile is currently addressing.

If the extension in question does not have an EXTNAME or HDUNAME keyword, then the function returns \$**HDU**\$n, where n is the sequential **HDU** index number (primary **HDU** = 0).

8.11.3.8 void CCfits::FITS::deleteExtension (int doomed)

Delete extension specified by extension number.

\overload

8.11.3.9 void CCfits::FITS::deleteExtension (const std::string & doomed, int version = 1)

Delete extension specified by name and version number.

Removes extension from **FITS** object and memory copy.

Parameters:

doomed the name of the extension to be deleted

version an optional version number, the EXTVER keyword, defaults to 1

Exceptions:

NoSuchHDU Thrown if there is no extension with the specified version number

FitsError Thrown if there is a non-zero status code from cfitsio, e.g. if the delete operation is applied to a **FITS** file opened for read only access.

8.11.3.10 void CCfits::FITS::destroy () throw ()

Erase **FITS** object and close corresponding file.

Force deallocation and erase of elements of a **FITS** memory object. Allows a reset of everything inside the **FITS** object, and closes the file. The object is inaccessible after this call.

destroy is public to allow users to reuse a symbol for a new file, but it is identical in operation to the destructor.

8.11.3.11 const std::multimap< string, ExtHdu *> & CCfits::FITS::extension<string,ExtHdu*> () const

return const reference to the extension container.

This is useful for such operations as `extension().size()` etc.

8.11.3.12 ExtHdu & CCfits::FITS::extension (const string & hduName, int version = 1) [inline]

return `FITS` extension by name and (optionally) version number.

8.11.3.13 const ExtHdu & CCfits::FITS::extension (const string & hduName, int version = 1) const

return `FITS` extension by name and (optionally) version number.

8.11.3.14 ExtHdu & CCfits::FITS::extension (int i)

return `FITS` extension by index number. non-const version. see const version for details.

8.11.3.15 const ExtHdu & CCfits::FITS::extension (int i) const

return `FITS` extension by index number. N.B. The input index number is currently defined as enumerating extensions, so the extension(1) returns `HDU` number 2.

8.11.3.16 Table & CCfits::FITS::filter (const string & expression, ExtHdu & inputTable, bool overwrite = true, bool readData = false)

Filter the rows of the `inputTable` with the condition expression, and return a reference to the resulting `Table`.

This function provides an object oriented version of cfitsio's `fits_select_rows` call. The expression string is any boolean expression involving the names of the columns in the input table (e.g., if there were a column called "density", a valid expression might be "DENSITY > 3.5": see the cfitsio documentation for further details).

[N.B. the "append" functionality described below does not work when linked with cfitsio 2.202 or prior because of a known issue with that version of the library. This causes the output to be a new extension with a correct header copy and version number but without the filtered data]. If the `inputTable` is an Extension `HDU` of this `FITS` object, then if `overwrite` is true the operation will overwrite the `inputTable` with the

filtered version, otherwise it will append a new **HDU** with the same extension name but the next highest version (EXTVER) number available.

8.11.3.17 void CCfits::FITS::flush ()

flush buffer contents to disk.

Provides manual control of disk writing operation. Image data are flushed automatically to disk after the write operation is completed, but not column data.

8.11.3.18 const string & CCfits::FITS::name () const

return filename of file corresponding to **FITS** object.

8.11.3.19 PHDU & CCfits::FITS::pHDU ()

return a reference to the primary **HDU**.

8.11.3.20 const PHDU & CCfits::FITS::pHDU () const

return a const reference to the primary **HDU**.

8.11.3.21 void CCfits::FITS::read (const std::vector< string > & searchKeys, const std::vector< string > & searchValues, bool readDataFlag = false, const std::vector< string > & hduKeys = std::vector<string>(), int version = 1)

read method for read header or **HDU** that contains specified keywords.

Parameters:

searchKeys A string vector of keywords to search for in each header

searchValues A string vector of values those keywords are required to have for success. Note that the keys must be of type string. If any value does not need to be checked the corresponding *searchValue* element can be empty.

readDataFlag boolean: if true, read data if **HDU** is found

hduKeys Allows optional reading of keys in the **HDU** that is searched for if it is successfully found

primaryKeys Allows optional reading of primary header keys on construction

version Optional version number. If specified, checks the EXTVERS keyword.

8.11.3.22 void CCfits::FITS::read (int hduIndex, bool readDataFlag = false, const std::vector< string > & keys = std::vector<string>())

read an **HDU** specified by index number.

This is provided to allow reading of HDUs after construction. see above for parameter definitions.

8.11.3.23 void CCfits::FITS::read (const std::vector< string > & *hduNames*, const std::vector< std::vector< string > > & *keys*, bool *readDataFlag* = false, const std::vector< int > & *hduVersions* = std::vector<int>())

get data from a set of HDUs from disk file, specifying keys and version numbers.

This is provided to allow reading of HDUs after construction. see above for parameter definitions.

8.11.3.24 void CCfits::FITS::read (const std::vector< string > & *hduNames*, bool *readDataFlag* = false)

get data from a set of HDUs from disk file.

This is provided to allow reading of HDUs after construction. see above for parameter definitions.

8.11.3.25 void CCfits::FITS::read (const string & *hduName*, bool *readDataFlag* = false, const std::vector< string > & *keys* = std::vector<string>(), int *version* = 0)

get data from single [HDU](#) from disk file.

This is provided to allow the adding of additional HDUs to the [FITS](#) object after construction of the [FITS](#) object. After the [read\(\)](#) functions have been called for the [FITS](#) object, subsequent read method to the Primary, [ExtHDU](#), and [Column](#) objects will retrieve data from the [FITS](#) object in memory (those methods can be called to read data in those [HDU](#) objects that was not read when the [HDU](#) objects were constructed).

All the read functions will throw [NoSuchHDU](#) exceptions on seek errors since they involve constructing [HDU](#) objects.

The parameter definitions are as documented for the corresponding constructor.

8.11.3.26 void CCfits::FITS::resetPosition ()

explicit call to set the fits file pointer to the primary.

8.11.3.27 void CCfits::FITS::setVerboseMode (bool *value*) [inline, static]

set verbose setting for library.

8.11.3.28 bool CCfits::FITS::verboseMode () [inline, static]

return verbose setting for library.

If true, all messages that are reported by exceptions are printed to std::cerr.

The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

8.12 CCfits::FitsError Class Reference

[FitsError](#) is the exception thrown by non-zero cfitsio status codes.

```
#include <FitsError.h>
```

Inheritance diagram for CCfits::FitsError::

Public Methods

- [FitsError](#) (int errornum, bool silent=true)

8.12.1 Detailed Description

[FitsError](#) is the exception thrown by non-zero cfitsio status codes.

8.12.2 Constructor & Destructor Documentation

8.12.2.1 CCfits::FitsError::FitsError (int *errornum*, bool *silent* = true)

ctor for cfitsio exception: translates status code into cfitsio error message.

The exception prefixes the string "Fits Error: " to the message printed by cfitsio.

Parameters:

errornum The cfitsio status code produced by the error.

silent A boolean controlling the printing of messages

The documentation for this class was generated from the following files:

- FitsError.h
- FitsError.cxx

8.13 CCfits::FitsException Class Reference

[FitsException](#) is the base class for all exceptions thrown by this library.

```
#include <FitsError.h>
```

Inheritance diagram for CCfits::FitsException::

Public Methods

- [FitsException](#) (const string &msg, bool &silent)

8.13.1 Detailed Description

[FitsException](#) is the base class for all exceptions thrown by this library.

All exceptions derived from this class can be caught by a single 'catch' clause catching [FitsException](#) by reference (which is the point of this base class design).

A static "verboseMode" parameter is provided by the [FITS](#) class to control diagnostics - if [FITS::verboseMode\(\)](#) is true, all diagnostics are printed (for debugging purposes). If not, then a boolean *silent* determines printing of messages. Each exception derived from [FitsException](#) must define a default value for the *silent* parameter.

8.13.2 Constructor & Destructor Documentation

8.13.2.1 CCfits::FitsException::FitsException (const string & msg, bool & silent)

Parameters:

diag A diagnostic string to be printed optionally.

silent A boolean controlling the printing of messages

The documentation for this class was generated from the following files:

- FitsError.h
- FitsError.cxx

8.14 CCfits::FitsFatal Class Reference

[potential] base class for exceptions to be thrown on internal library error.

```
#include <FitsError.h>
```

Public Methods

- [FitsFatal](#) (const string &diag)

8.14.1 Detailed Description

[potential] base class for exceptions to be thrown on internal library error.

As of this version there are no subclasses. This error requests that the user reports this circumstance to HEASARC.

8.14.2 Constructor & Destructor Documentation

8.14.2.1 CCfits::FitsFatal::FitsFatal (const string & diag)

Prints a message starting "*** CCfits Fatal Error: ..." and calls *terminate()*.

Parameters:

diag A diagnostic string to be printed identifying the context of the error.

The documentation for this class was generated from the following files:

- FitsError.h
- FitsError.cxx

8.15 CCfits::HDU Class Reference

Base class for all [HDU](#) [Header-Data Unit] objects.

```
#include <HDU.h>
```

Inheritance diagram for CCfits::HDU::

Public Methods

- [HDU](#) (const HDU &right)
- bool [operator==](#) (const HDU &right) const
- bool [operator!=](#) (const HDU &right) const
- virtual HDU* [clone](#) (FITSBase *p) const=0
- fitsfile* [fitsPointer](#) () const
- virtual void [makeThisCurrent](#) () const

- std::map<string, [Keyword](#)*> [keyWord](#) ()
- [Keyword& keyWord](#) (const string &keyName)
- const string& [getHistory](#) ()
- const string& [getComments](#) ()
- void [writeHistory](#) (const string &history="Generic History String")
- void [writeComment](#) (const string &comment="Generic Comment")
- void [writeDate](#) ()
- FITSBase* [parent](#) () const
- void [readAllKeys](#) ()
- long [axes](#) () const
- long [axis](#) (size_t index) const
- void [index](#) (int value)
- void [deleteKey](#) (const std::string &doomed)
- long [bitpix](#) () const
- void [bitpix](#) (long value)
- int [index](#) () const
- const string& [history](#) () const
- const string& [comment](#) () const
- virtual double [zero](#) () const
- virtual double [scale](#) () const
- const std::map<string, [Keyword](#)*>& [keyWord](#) () const
- const [Keyword& keyWord](#) (const string &keyname) const
- void [setKeyWord](#) (const string &keyname, [Keyword](#) &value)
- template<typename T> [Keyword& addKey](#) (const string &name, T val, const string &comment)
- template<typename T> void [readKey](#) (const string &keyName, T &val)
- template<typename T> void [readKeys](#) (std::vector< string > &keyNames, std::vector< T > &vals)

Protected Methods

- [HDU](#) (FITSBase *p=0)
- [HDU](#) (FITSBase *p, int bitpix, int naxis, const std::vector< long > &axes)
- virtual [~HDU](#) ()
- std::vector< long >& [naxes](#) ()

8.15.1 Detailed Description

Base class for all [HDU](#) [Header-Data Unit] objects.

[HDU](#) objects in CCfits are either [PHDU](#) (Primary [HDU](#) objects) or [ExtHDU](#) (Extension [HDU](#)) objects. Following the behavior. ExtHDUs are further subclassed into [ImageExt](#) or [Table](#) objects, which are finally [AsciiTable](#) or [BinTable](#) objects.

`HDU`'s public interface gives access to properties that are common to all HDUs, largely required keywords, and functions that are common to all HDUs, principally the manipulation of keywords and their values.

HDUs must be constructed by `HDUCreator` objects which are called by `FITS` methods. Each `HDU` has an embedded pointer to a `FITSBase` object, which is private to `FITS` [`FITSBase` is a pointer encapsulating the resources of `FITS`. For details of this coding idiom see *Exceptional C++* by Herb Sutter (2000) and references therein].

8.15.2 Constructor & Destructor Documentation

8.15.2.1 CCfits::HDU::HDU (`const HDU & right`)

copy constructor.

8.15.2.2 CCfits::HDU::HDU (`FITSBase * p = 0`) [protected]

default constructor, called by `HDU` subclasses that read from `FITS` files.

8.15.2.3 CCfits::HDU::HDU (`FITSBase * p, int bitpix, int naxis, const std::vector< long > & axes`) [protected]

constructor for creating new `HDU` objects, called by `HDU` subclasses writing to `FITS` files.

8.15.2.4 CCfits::HDU::~HDU () [protected, virtual]

destructor.

8.15.3 Member Function Documentation

8.15.3.1 template<typename T> `Keyword &` CCfits::HDU::addKey (`const string & name, T value, const string & comment`)

create a new keyword in the `HDU` with specified value and comment fields.

the function returns a reference to keyword object just created.

Parameters:

Parameters:

`name` (`std::string`) The keyword name

`value` (`T = std::string, float, std::complex<float>, int, or bool`)

`comment` (`std::string`) the keyword value

Note the limitations on T, which arise for historical reasons (i.e. since keywords do not contain explicit type information, routines in CCfits must assign a type). The limitation may be removed in a future release

It is possible however to create a keyword of any of the allowed data types in fitsio (see the cfitsio manual section 4.3). The required calls are, e.g. for a floating point double valued keyword:

```
NewKeyword<double> keyCreate( hdu, double val1 ); Keyword* key1 = Keyword keyCreate.MakeKeyword(keyName1, const string& comment); keyCreate.keyData(val2); Keyword* key2 = Keyword keyCreate.MakeKeyword(keyName2, const string& comment);  
key1->write(); key2->write();  
... which is essentially what addKey does for its allowed types.
```

8.15.3.2 long CCfits::HDU::axes () const [inline]

return the number of axes in the **HDU** data section (always 2 for tables).

8.15.3.3 long CCfits::HDU::axis (size_t index) const [inline]

return the length of **HDU** data axis i.

8.15.3.4 void CCfits::HDU::bitpix (long value) [inline]

set the data type keyword.

8.15.3.5 long CCfits::HDU::bitpix () const [inline]

return the data type keyword.

Takes values denoting the image data type for images, and takes the fixed value 8 for tables.

8.15.3.6 HDU * CCfits::HDU::clone (FITSBase * p) const [pure virtual]

virtual copy constructor, to be implemented in subclasses.

Reimplemented in **CCfits::AsciiTable**, **CCfits::BinTable**, **CCfits::ExtHDU**, and **CCfits::PHDU**.

8.15.3.7 const string & CCfits::HDU::comment () const [inline]

return the comment string previously read by getComment().

8.15.3.8 void CCfits::HDU::deleteKey (const std::string & *doomed*)

delete a keyword from the header.

removes *doomed* from the [FITS](#) file and from the [FITS](#) object

8.15.3.9 fitsfile * CCfits::HDU::fitsPointer () const

return the fitsfile pointer for the [FITS](#) object containing the [HDU](#).

8.15.3.10 const string & CCfits::HDU::getComments ()

read the comments from the [HDU](#) and add it to the [FITS](#) object.

The comment string found in the header is concatenated and returned to the calling function

8.15.3.11 const string & CCfits::HDU::getHistory ()

read the history information from the [HDU](#) and add it to the [FITS](#) object.

The history string found in the header is concatenated and returned to the calling function

8.15.3.12 const string & CCfits::HDU::history () const [inline]

return the history string previously read by [getHistory\(\)](#).

8.15.3.13 int CCfits::HDU::index () const [inline]

return the [HDU](#) number.

8.15.3.14 void CCfits::HDU::index (int *value*) [inline]

set the [HDU](#) number.

8.15.3.15 const [Keyword](#) & CCfits::HDU::keyWord (const string & *keyname*) const [inline]

return a (previously read) keyword from the [HDU](#) object. const version.

8.15.3.16 const std::map< string, [Keyword](#) *> & CCfits::HDU::keyWord () const [inline]

return the associative array containing the [HDU](#) Keywords that have been read so far.

8.15.3.17 `Keyword & CCfits::HDU::keyWord (const string & keyName)`
[inline]

return a (previously read) keyword from the [HDU](#) object.

8.15.3.18 `std::map< string, Keyword *> CCfits::HDU::keyWord ()`
[inline]

return the associative array containing the [HDU](#) keywords so far read.

8.15.3.19 `void CCfits::HDU::makeThisCurrent () const [virtual]`

move the fitsfile pointer to this current [HDU](#).

This function should never need to be called by the user since it is called internally whenever required.

Reimplemented in [CCfits::ExtHDU](#).

8.15.3.20 `std::vector< long > & CCfits::HDU::naxes () [inline, protected]`

return the [HDU](#) data axis array.

8.15.3.21 `bool CCfits::HDU::operator!= (const HDU & right) const`

inequality operator.

8.15.3.22 `bool CCfits::HDU::operator== (const HDU & right) const`

equality operator.

8.15.3.23 `FITSBase * CCfits::HDU::parent () const`

return reference to the pointer representing the FITSBase object containing the [HDU](#).

8.15.3.24 `void CCfits::HDU::readAllKeys ()`

read all of the keys in the header.

This member function reads keys that are not meta data for columns or image information, [which are considered to be part of the column or image objects]. Also, history and comment keys are read and returned by [getHistory\(\)](#) and [getComment\(\)](#).

Note that [readAllKeys](#) can only construct keys of type string, float, complex<float>, integer, and bool because the [FITS](#) header records do not encode exact type information.

8.15.3.25 template<typename T> void CCfits::HDU::readKey (const string & keyName, T & val)

read a keyword of specified type from the header of a disk [FITS](#) file and return its value.
T is one of the types std::string, float, int, std::complex<float>, and bool.

8.15.3.26 template<typename T> void CCfits::HDU::readKeys (std::vector<string > & keyNames, std::vector< T > & vals)

read a set of specified keywords of the same data type from the header of a disk [FITS](#) file and return their values.

T is one of the types std::string, float, int, std::complex<float>, and bool.

8.15.3.27 double CCfits::HDU::scale () const [inline, virtual]

return the BSCALE keyword value.

Reimplemented in [CCfits::PHDU](#).

8.15.3.28 void CCfits::HDU::setKeyWord (const string & keyname, Keyword & value) [inline]

set [Keyword](#) keyname to specified value.

8.15.3.29 void CCfits::HDU::writeComment (const string & comment = "Generic Comment")

write a comment string.

A default value for the string is given ("Generic Comment String") so users can put a placeholder call to this function in their code.

8.15.3.30 void CCfits::HDU::writeDate ()

write a date string to *this.

8.15.3.31 void CCfits::HDU::writeHistory (const string & history = "Generic History String")

write a history string.

A default value for the string is given ("Generic History String") so users can put a placeholder call to this function in their code.

8.15.3.32 double CCfits::HDU::zero () const [inline, virtual]

return the BZERO keyword value.

Reimplemented in [CCfits::PHDU](#).

The documentation for this class was generated from the following files:

- HDU.h
- FITS.h
- HDU.cxx

8.16 CCfits::Column::InsufficientElements Class Reference

Exception thrown if the data supplied for a write operation is less than declared.

```
#include <Column.h>
```

Inheritance diagram for CCfits::Column::InsufficientElements::

Public Methods

- [InsufficientElements](#) (const string &msg, bool silent=true)

8.16.1 Detailed Description

Exception thrown if the data supplied for a write operation is less than declared.

This circumstance generates an exception to avoid unexpected behaviour after the write operation is completed. It can be avoided by resizing the input array appropriately.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 CCfits::Column::InsufficientElements::InsufficientElements (const string & msg, bool silent = true)

Exception ctor, prefixes the string "[FitsError](#): not enough elements supplied for write operation: " before the specific message.

Parameters:

diag A specific diagnostic message, usually the column name

silent if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

8.17 CCfits::Column::InvalidDataType Class Reference

Exception thrown for invalid data type inputs.

```
#include <Column.h>
```

Inheritance diagram for CCfits::Column::InvalidDataType::

Public Methods

- [InvalidDataType \(const string &str=""", bool silent=true\)](#)

8.17.1 Detailed Description

Exception thrown for invalid data type inputs.

This exception is thrown if the user requests an implicit data type conversion to a datatype that is not one of the supported types (see fitsio.h for details).

8.17.2 Constructor & Destructor Documentation

8.17.2.1 CCfits::Column::InvalidDataType::InvalidDataType (const string & str = "", bool silent = true)

Exception ctor, prefixes the string "[FitsError](#): Incorrect data type: " before the specific message.

Parameters:

- str* A specific diagnostic message
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

8.18 CCfits::HDU::InvalidExtensionType Class Reference

exception to be thrown if user requests extension type that can not be understood as `ImageExt`, `AsciiTable` or `BinTable`.

```
#include <HDU.h>
```

Inheritance diagram for CCfits::HDU::InvalidExtensionType::

Public Methods

- `InvalidExtensionType (const string &diag, bool silent=true)`

8.18.1 Detailed Description

exception to be thrown if user requests extension type that can not be understood as `ImageExt`, `AsciiTable` or `BinTable`.

8.18.2 Constructor & Destructor Documentation

8.18.2.1 CCfits::HDU::InvalidExtensionType::InvalidExtensionType (const string & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "Fits Error: Extension Type: " before the specific message.

Parameters:

- diag* A specific diagnostic message
- silent* if true, print message whether `FITS::verboseMode` is set or not.

The documentation for this class was generated from the following files:

- `HDU.h`
- `HDU.cxx`

8.19 CCfits::HDU::InvalidImageDataType Class Reference

exception to be thrown if user requests creation of an image of type not supported by cfitsio.

```
#include <HDU.h>
```

Inheritance diagram for CCfits::HDU::InvalidImageDataType::

Public Methods

- [InvalidImageDataType \(const string &diag, bool silent=true\)](#)

8.19.1 Detailed Description

exception to be thrown if user requests creation of an image of type not supported by cfitsio.

8.19.2 Constructor & Destructor Documentation

8.19.2.1 CCfits::HDU::InvalidImageDataType::InvalidImageDataType (const string & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "Fits Error: Invalid Data Type for Image " before the specific message.

Parameters:

- diag* A specific diagnostic message
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- HDU.h
- HDU.cxx

8.20 CCfits::Column::InvalidNumberOfRows Class Reference

Exception thrown if user enters a non-positive number for the number of rows to write.

```
#include <Column.h>
```

Inheritance diagram for CCfits::Column::InvalidNumberOfRows::

Public Methods

- [InvalidNumberOfRows \(size_t number, bool silent=true\)](#)

8.20.1 Detailed Description

Exception thrown if user enters a non-positive number for the number of rows to write.

8.20.2 Constructor & Destructor Documentation

8.20.2.1 CCfits::Column::InvalidNumberOfRows::InvalidNumberOfRows (*size_t number*, *bool silent = true*)

Exception ctor, prefixes the string "Fits Error: number of rows to write must be positive" before the specific message.

Parameters:

- *number* The number of rows entered.
- *silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

8.21 CCfits::Column::InvalidRowNumber Class Reference

Exception thrown on attempting to read a row number beyond the end of a table.

```
#include <Column.h>
```

Inheritance diagram for CCfits::Column::InvalidRowNumber::

Public Methods

- [InvalidRowNumber](#) (const string &diag, bool silent=true)

8.21.1 Detailed Description

Exception thrown on attempting to read a row number beyond the end of a table.

8.21.2 Constructor & Destructor Documentation

8.21.2.1 CCfits::Column::InvalidRowNumber::InvalidRowNumber (const string & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "FitsError: Invalid Row Number - Column: " before the specific message.

Parameters:

- *diag* A specific diagnostic message, usually the column name.
- *silent* if true, print message whether FITS::verboseMode is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

8.22 CCfits::Column::InvalidRowParameter Class Reference

Exception thrown on incorrect row writing request.

```
#include <Column.h>
```

Inheritance diagram for CCfits::Column::InvalidRowParameter::

Public Methods

- [InvalidRowParameter \(const string &diag, bool silent=true\)](#)

8.22.1 Detailed Description

Exception thrown on incorrect row writing request.

This exception is thrown if the user requests writing more data than a fixed width row can accommodate. An exception is thrown rather than a truncation because it is likely that the user will not otherwise realize that data loss is happening.

8.22.2 Constructor & Destructor Documentation

8.22.2.1 CCfits::Column::InvalidRowParameter::InvalidRowParameter (const string & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "[FitsError](#): row offset or length incompatible with column declaration " before the specific message.

Parameters:

- *diag* A specific diagnostic message, usually the column name
- *silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

8.23 CCfits::Keyword Class Reference

Abstract base class implementing the common behavior of [Keyword](#) objects.

```
#include <Keyword.h>
```

Public Methods

- [Keyword](#) (const [Keyword](#) &right)
- [Keyword](#) (const string &keyname, [ValueType](#) keytype, [HDU](#) *p, const string &comment=""")
- virtual [~Keyword](#) ()
- [Keyword& operator=](#) (const [Keyword](#) &right)
- bool [operator==](#) (const [Keyword](#) &right) const
- bool [operator!=](#) (const [Keyword](#) &right) const
- virtual [Keyword*](#) [clone](#) () const=0
- virtual void [write](#) ()=0
- [fitsfile*](#) [fitsPointer](#) () const
- const std::string& [name](#) () const
- const std::string& [comment](#) () const

Protected Methods

- [ValueType](#) [keytype](#) () const
- void [keytype](#) ([ValueType](#) value)
- const [HDU*](#) [parent](#) () const

8.23.1 Detailed Description

Abstract base class implementing the common behavior of [Keyword](#) objects.

Keywords consists of a name, a value and a comment field. Concrete templated subclasses, [KeyData](#)<T>, have a data member that holds the value of keyword.

Typically, the mandatory keywords for a given [HDU](#) type are not stored as object of type [Keyword](#), but as intrinsic data types. The [Keyword](#) hierarchy is used to store user-supplied information.

8.23.2 Constructor & Destructor Documentation

8.23.2.1 CCfits::Keyword::Keyword (const [Keyword](#) & *right*)

copy constructor.

8.23.2.2 CCfits::Keyword::Keyword (const string & *keyname*, [ValueType](#) *keytype*, [HDU](#) * *p*, const string & *comment* = "")

[Keyword](#) constructor.

This is the common behavior of Keywords of any type. Constructor is protected as the class is abstract.

8.23.2.3 CCfits::Keyword::~Keyword () [virtual]

virtual destructor.

8.23.3 Member Function Documentation

8.23.3.1 [Keyword](#) * CCfits::Keyword::clone () const [pure virtual]

virtual copy constructor.

8.23.3.2 const std::string & CCfits::Keyword::comment () const [inline]

return the comment field of the keyword.

8.23.3.3 fitsfile * CCfits::Keyword::fitsPointer () const

return a pointer to the [FITS](#) file containing the parent [HDU](#).

8.23.3.4 void CCfits::Keyword::keytype ([ValueType](#) *value*) [inline, protected]

set keyword type.

8.23.3.5 ValueType CCfits::Keyword::keytype () const [inline, protected]

return the type of a keyword.

8.23.3.6 const std::string & CCfits::Keyword::name () const [inline]

return the name of a keyword

8.23.3.7 bool CCfits::Keyword::operator!= (const Keyword & right) const

inequality operator.

8.23.3.8 Keyword & CCfits::Keyword::operator= (const Keyword & right)

assignment operator.

8.23.3.9 bool CCfits::Keyword::operator== (const Keyword & right) const

equality operator.

8.23.3.10 const HDU * CCfits::Keyword::parent () const [inline, protected]

return a pointer to parent **HDU**.

8.23.3.11 void CCfits::Keyword::write () [pure virtual]

write operation.

The documentation for this class was generated from the following files:

- [Keyword.h](#)
- [HDU.h](#)
- [Keyword.cxx](#)
- [KeywordT.h](#)

8.24 CCfits::FITSUtil::MatchName Class Template Reference

predicate for classes that have a name attribute; match input string with instance name.

```
#include <FITSUtil.h>
```

8.24.1 Detailed Description

template<class T> class CCfits::FITSUtil::MatchName

predicate for classes that have a name attribute; match input string with instance name.

Usage: **MatchName<NamedClass> Ex;**

list<NamedClass> ListObject;

... ...

find_if(ListObject.begin(),ListObject().end(),bind2nd(Ex,"needle"));

Since most of the classes within CCfits are not implemented with lists, these functions are now of little direct use.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.25 CCfits::FITSUtil::MatchNum Class Template Reference

predicate for classes that have an index attribute; match input index with instance value.

#include <FITSUtil.h>

8.25.1 Detailed Description

template<class T> class CCfits::FITSUtil::MatchNum

predicate for classes that have an index attribute; match input index with instance value.

Usage: **MatchName<IndexedClass> Ex;**

list<NamedClass> ListObject;

... ...

find_if(ListObject.begin(),ListObject().end(),bind2nd(Ex,5));

Since most of the classes within CCfits are implemented with std::maps rather than lists, these functions are now of little direct use.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.26 CCfits::FITSUtil::MatchPtrName Class Template Reference

as for [MatchName](#), only with the input class a pointer.

```
#include <FITSUtil.h>
```

8.26.1 Detailed Description

template<class T> class CCfits::FITSUtil::MatchPtrName

as for [MatchName](#), only with the input class a pointer.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.27 CCfits::FITSUtil::MatchPtrNum Class Template Reference

as for [MatchNum](#), only with the input class a pointer.

```
#include <FITSUtil.h>
```

8.27.1 Detailed Description

template<class T> class CCfits::FITSUtil::MatchPtrNum

as for [MatchNum](#), only with the input class a pointer.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.28 CCfits::FITSUtil::MatchType Class Template Reference

function object that returns the [FITS](#) ValueType corresponding to an input intrinsic type.

```
#include <FITSUtil.h>
```

8.28.1 Detailed Description

template<typename T> class CCfits::FITSUtil::MatchType

function object that returns the **FITS** ValueType corresponding to an input intrinsic type.

This is particularly useful inside templated class instances where calls to cfitsio need to supply a value type. With this function one can extract the value type from the class type.

usage:

MatchType<*T*> *type*;

ValueType *dataType* = *type*();

Uses run-time type information (RTTI) methods.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.29 CCfits::HDU::NoNullValue Class Reference

exception to be thrown on seek errors for keywords.

```
#include <HDU.h>
```

Inheritance diagram for CCfits::HDU::NoNullValue::

Public Methods

- **NoNullValue** (const string &diag, bool silent=true)

8.29.1 Detailed Description

exception to be thrown on seek errors for keywords.

8.29.2 Constructor & Destructor Documentation

8.29.2.1 CCfits::HDU::NoNullValue::NoNullValue (const string & diag, bool silent = true)

Exception ctor, prefixes the string "Fits Error: No Null Pixel Value specified for Image" before the specific message.

Parameters:

- *diag* A specific diagnostic message, the name of the **HDU** if not the primary.
- *silent* if true, print message whether **FITS::verboseMode** is set or not.

The documentation for this class was generated from the following files:

- HDU.h
- HDU.cxx

8.30 CCfits::Column::NoNullValue Class Reference

Exception thrown if a null value is specified without support from existing column header.

```
#include <Column.h>
```

Inheritance diagram for CCfits::Column::NoNullValue::

Public Methods

- **NoNullValue** (const string &*diag*, bool *silent*=true)

8.30.1 Detailed Description

Exception thrown if a null value is specified without support from existing column header.

This exception is analogous to the fact that cfitsio returns a non-zero status code if TNULLn doesn't exist an a null value (convert all input data with the null value to the TNULLn keyword) is specified. It is only relevant for integer type data (see cfitsio manual for details).

8.30.2 Constructor & Destructor Documentation

8.30.2.1 CCfits::Column::NoNullValue::NoNullValue (const string & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "Fits Error: No null value specified for column: " before the specific message.

Parameters:

- *diag* A specific diagnostic message
- *silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

8.31 CCfits::Table::NoSuchColumn Class Reference

Exception to be thrown on a failure to retrieve a column specified either by name or index number.

```
#include <Table.h>
```

Inheritance diagram for CCfits::Table::NoSuchColumn::

Public Methods

- [NoSuchColumn](#) (const string &name, bool silent=true)
- [NoSuchColumn](#) (int index, bool silent=true)

8.31.1 Detailed Description

Exception to be thrown on a failure to retrieve a column specified either by name or index number.

When a [Table](#) object is created, the header is read and a column object created for each column defined. Thus if this exception is thrown the column requested does not exist in the [HDU](#) (note that the column can easily exist and not contain any data since the user controls whether the column will be read when the [FITS](#) object is instantiated).

It is expected that the index number calls will be primarily internal. The underlying implementation makes lookup by name more efficient.

The exception has two variants, which take either an integer or a string as parameter. These are used according to the accessor that threw them, either by name or index.

8.31.2 Constructor & Destructor Documentation

8.31.2.1 CCfits::Table::NoSuchColumn::NoSuchColumn (const string & *name*, bool *silent* = true)

Exception ctor for exception thrown if the requested column (specified by name) is not present.

Message: Fits Error: cannot find [Column](#) named: *name* is printed.

Parameters:

name the requested column name

silent if true, print message whether [FITS::verboseMode](#) is set or not.

8.31.2.2 CCfits::Table::NoSuchColumn::NoSuchColumn (int *index*, bool *silent* = true)

Exception ctor for exception thrown if the requested column (specified by name) is not present.

Message: Fits Error: column not present - [Column](#) number *index* is printed.

Parameters:

index the requested column number

silent if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Table.h
- Table.cxx

8.32 CCfits::FITS::NoSuchHDU Class Reference

exception thrown by [HDU](#) retrieval methods.

```
#include <FITS.h>
```

Inheritance diagram for CCfits::FITS::NoSuchHDU::

Public Methods

- [NoSuchHDU](#) (const string &diag, bool silent=true)

8.32.1 Detailed Description

exception thrown by [HDU](#) retrieval methods.

8.32.2 Constructor & Destructor Documentation

8.32.2.1 CCfits::FITS::NoSuchHDU::NoSuchHDU (const string & *diag*, bool *silent* = **true**)

Exception ctor, prefixes the string "FITS Error: Cannot read **HDU** in **FITS** file:" before the specific message.

Parameters:

- *diag* A specific diagnostic message, usually the name of the extension whose read was attempted.
- *silent* if true, print message whether **FITS::verboseMode** is set or not.

The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

8.33 CCfits::HDU::NoSuchKeyword Class Reference

exception to be thrown on seek errors for keywords.

```
#include <HDU.h>
```

Inheritance diagram for CCfits::HDU::NoSuchKeyword::

Public Methods

- [NoSuchKeyword](#) (const string &*diag*, bool *silent*=**true**)

8.33.1 Detailed Description

exception to be thrown on seek errors for keywords.

8.33.2 Constructor & Destructor Documentation

8.33.2.1 CCfits::HDU::NoSuchKeyword::NoSuchKeyword (const string & *diag*, bool *silent* = **true**)

Exception ctor, prefixes the string "Fits Error: **Keyword** not found: " before the specific message.

Parameters:

diag A specific diagnostic message, usually the name of the keyword requested.

silent if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- [HDU.h](#)
- [HDU.cxx](#)

8.34 CCfits::FITS::OperationNotSupported Class Reference

thrown for unsupported operations, such as attempted to select rows from an image extension.

```
#include <FITS.h>
```

Inheritance diagram for CCfits::FITS::OperationNotSupported::

Public Methods

- [OperationNotSupported \(const string &msg, bool silent=true\)](#)

8.34.1 Detailed Description

thrown for unsupported operations, such as attempted to select rows from an image extension.

8.34.2 Constructor & Destructor Documentation

8.34.2.1 CCfits::FITS::OperationNotSupported::OperationNotSupported (const string & msg, bool silent = true)

Exception ctor, prefixes the string "FITS Error: Operation not supported:" before the specific message.

Parameters:

msg A specific diagnostic message.

silent if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- [FITS.h](#)
- [FITS.cxx](#)

8.35 CCfits::PHDU Class Reference

class representing the primary [HDU](#) for a [FITS](#) file.

```
#include <PHDU.h>
```

Inheritance diagram for CCfits::PHDU::

Public Methods

- virtual PHDU* [clone](#) (FITSBase *p) const=0
- virtual void [zero](#) (double value)
- virtual double [zero](#) () const
- virtual double [scale](#) () const
- template<typename S> void [write](#) (const std::vector< long > &first, long nElements, const std::valarray< S > &data, S *nullValue)
- template<typename S> void [write](#) (long first, long nElements, const std::valarray< S > &data, S *nullValue)
- template<typename S> void [write](#) (const std::vector< long > &first, long nElements, const std::valarray< S > &data)
- template<typename S> void [write](#) (long first, long nElements, const std::valarray< S > &data)
- template<typename S> void [write](#) (const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::valarray< S > &data)
- template<typename S> void [read](#) (std::valarray< S > &image, long first, long nElements)
- template<typename S> void [read](#) (std::valarray< S > &image, long first, long nElements, S *nullValue)
- template<typename S> void [read](#) (std::valarray< S > &image, const std::vector< long > &first, long nElements)
- template<typename S> void [read](#) (std::valarray< S > &image, const std::vector< long > &first, long nElements, S *nullValue)
- template<typename S> void [read](#) (std::valarray< S > &image, const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::vector< long > &stride)
- template<typename S> void [read](#) (std::valarray< S > &image, const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::vector< long > &stride, S *nullValue)

Protected Methods

- [PHDU](#) (const PHDU &right)
- [PHDU](#) (FITSBase *p, int bpix, int naxis, const std::vector< long > &axes)
- [PHDU](#) (FITSBase *p=0)

8.35.1 Detailed Description

class representing the primary [HDU](#) for a [FITS](#) file.

A [PHDU](#) object is automatically instantiated and added to a [FITS](#) object when a [FITS](#) file is accessed in any way. If a new file is created without specifying the data type for the header, CCfits assumes that the file is to be used for table extensions and creates a dummy header. [PHDU](#) instances are *only* created by [FITS](#) ctors. In the first release of CCfits, the Primary cannot be changed once declared.

[PHDU](#) and [ExtHDU](#) provide the same interface to writing images: multiple overloads of the templated [PHDU::read](#) and [PHDU::write](#) operations provide for (a) writing image data specified in a number of ways [C-array, std::vector, std::valarray] and with input location specified by initial pixel, by n-tuple, and by rectangular subset [generalized slice]; (b) reading image data specified similarly to the write options into a std::valarray.

Todo:

Implement functions that allow replacement of the primary image

8.35.2 Constructor & Destructor Documentation

8.35.2.1 CCfits::PHDU::PHDU (const PHDU & right) [protected]

copy constructor.

required for cloning primary HDUs when copying [FITS](#) files.

8.35.2.2 CCfits::PHDU::PHDU (FITSB_{ase} * p, int bpix, int naxis, const std::vector< long > & axes) [protected]

Writing Primary [HDU](#) constructor, called by PrimaryHDU<T> class.

Constructor used for creating new [PHDU](#) (i.e. for writing data to [FITS](#)). also doubles as default constructor since all arguments have default values, which are passed to the [HDU](#) constructor

8.35.2.3 CCfits::PHDU::PHDU (FITSB_{ase} * p = 0) [protected]

Reading Primary [HDU](#) constructor.

Constructor used when reading the primary [HDU](#) from an existing file. Does nothing except initialize, with the real work done by the subclass PrimaryHDU<T>.

8.35.3 Member Function Documentation

8.35.3.1 PHDU * CCfits::PHDU::clone (FITSB_{ase} * p) const [pure virtual]

virtual copy constructor, to be implemented in subclasses.

Reimplemented from [CCfits::HDU](#).

8.35.3.2 template<typename S> void CCfits::PHDU::read (std::valarray< S > & *image*, const std::vector< long > & *firstVertex*, const std::vector< long > & *lastVertex*, const std::vector< long > & *stride*, S * *nullValue*)

read an image subset into valarray image, processing null values.

The image subset is defined by two vertices and a stride indicating the 'densemess' of the values to be picked in each dimension (a stride = (1,1,1,...) means picking every pixel in every dimension, whereas stride = (2,2,2,...) means picking every other value in each dimension.

8.35.3.3 template<typename S> void CCfits::PHDU::read (std::valarray< S > & *image*, const std::vector< long > & *firstVertex*, const std::vector< long > & *lastVertex*, const std::vector< long > & *stride*)

read an image subset.

8.35.3.4 template<typename S> void CCfits::PHDU::read (std::valarray< S > & *image*, const std::vector< long > & *first*, long *nElements*, S * *nullValue*)

read part of an image array, processing null values.

As above except for

Parameters:

first a vector<long> representing an n-tuple giving the coordinates in the image of the first pixel.

8.35.3.5 template<typename S> void CCfits::PHDU::read (std::valarray< S > & *image*, const std::vector< long > & *first*, long *nElements*)

read an image section starting at a location specified by an n-tuple.

8.35.3.6 template<typename S> void CCfits::PHDU::read (std::valarray< S > & *image*, long *first*, long *nElements*, S * *nullValue*)

read part of an image array, processing null values.

Implicit data conversion is supported (i.e. user does not need to know the type of the data stored. A WrongExtensionType extension is thrown if *this is not an image.

Parameters:

image The receiving container, a std::valarray reference

first The first pixel from the array to read [a long value]

nElements The number of values to read

nullValue A pointer containing the value in the table to be considered as undefined. See cfitsio for details

8.35.3.7 template<typename S> void CCfits::PHDU::read (std::valarray< S > & *image*, long *first*, long *nElements*)

read an image section starting at a specified pixel.

8.35.3.8 double CCfits::PHDU::scale () const [virtual]

return the BSCALE keyword value.

Reimplemented from [CCfits::HDU](#).

8.35.3.9 template<typename S> void CCfits::PHDU::write (const std::vector< long > & *firstVertex*, const std::vector< long > & *lastVertex*, const std::valarray< S > & *data*)

write a subset (generalize slice) of data to the image.

A generalized slice/subset is a subset of the image (e.g. one plane of a data cube of size \leq the dimension of the cube). It is specified by two opposite vertices. The equivalent cfitsio call does not support undefined data processing so there is no version that allows a null value to be specified.

Parameters:

firstVertex the coordinates specifying lower and upper vertices of the n-dimensional slice

lastVertex

data The data to be written

8.35.3.10 template<typename S> void CCfits::PHDU::write (long *first*, long *n-Elements*, const std::valarray< S > & *data*)

write array starting from specified pixel number, without undefined data processing.

8.35.3.11 template<typename S> void CCfits::PHDU::write (const std::vector< long > & *first*, long *nElements*, const std::valarray< S > & *data*)

write array starting from specified n-tuple, without undefined data processing.

8.35.3.12 template<typename S> void CCfits::PHDU::write (long *first*, long *nElements*, const std::valarray< S > & *data*, S * *nullValue*)

write array to image starting with a specified pixel and allowing undefined data to be processed.

parameters after the first are as for version with n-tuple specifying first element. these two version are equivalent, except that it is possible for the first pixel number to exceed the range of 32-bit integers, which is how long datatype is commonly implemented.

8.35.3.13 template<typename S> void CCfits::PHDU::write (const std::vector< long > & *first*, long *nElements*, const std::valarray< S > & *data*, S * *nullValue*)

Write a set of pixels to an image extension with the first pixel specified by an n-tuple, processing undefined data.

All the overloaded versions of [PHDU::write](#) perform operations on **this* if it is an image and throw a WrongExtensionType exception if not. Where appropriate, alternate versions allow undefined data to be processed

Parameters:

- first* an n-tuple of dimension equal to the image dimension specifying the first pixel in the range to be written
- nElements* number of pixels to be written
- data* array of data to be written
- pointer* to null value (data with this value written as undefined; needs the BLANK keyword to have been specified).

8.35.3.14 double CCfits::PHDU::zero () const [virtual]

return the BZERO keyword value.

Reimplemented from [CCfits::HDU](#).

8.35.3.15 void CCfits::PHDU::zero (double *value*) [virtual]

Private: called by ctor.

Reimplemented from [CCfits::HDU](#).

The documentation for this class was generated from the following files:

- PHDU.h
- PHDU.cxx
- PHDUT.h

8.36 CCfits::Column::RangeError Class Reference

exception to be thrown for inputs that cause range errors in column read operations.

```
#include <Column.h>
```

Inheritance diagram for CCfits::Column::RangeError::

Public Methods

- [RangeError](#) (const string &msg, bool silent=true)

8.36.1 Detailed Description

exception to be thrown for inputs that cause range errors in column read operations.

Range errors here mean (last < first) in a request to read a range of rows.

8.36.2 Constructor & Destructor Documentation

8.36.2.1 CCfits::Column::RangeError::RangeError (const string & msg, bool silent = true)

Exception ctor, prefixes the string "[FitsError](#): Range error in operation " before the specific message.

Parameters:

- msg* A specific diagnostic message
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

8.37 CCfits::Table Class Reference

```
#include <Table.h>
```

Inheritance diagram for CCfits::Table::

Public Methods

- [Table \(const Table &right\)](#)
- virtual [~Table \(\)](#)
- const std::map<string, Column*>& [column \(\) const](#)
- virtual [Column& column \(const string &colName\) const](#)
- virtual [Column& column \(int colIndex\) const](#)
- virtual long [rows \(\) const](#)
- void [updateRows \(\)](#)
- void [rows \(long numRows\)](#)
- virtual void [deleteColumn \(const string &name\)](#)
- void [insertRows \(long first, long number=1\)](#)
- void [deleteRows \(long first, long number=1\)](#)
- void [deleteRows \(const std::vector< long > &rowList\)](#)
- virtual std::map<string, Column*>& [column \(\)](#)

Protected Methods

- [Table \(FITSBase *p, HduType xtype, const string &hduName, int rows, const std::vector< string > &columnName, const std::vector< string > &columnFmt, const std::vector< string > &columnUnit=std::vector< string >\(\), int version=1\)](#)
- [Table \(FITSBase *p, HduType xtype, const string &hduName=string\(""\)\), int version=1\)](#)
- [Table \(FITSBase *p, HduType xtype, int number\)](#)
- void [init \(bool readFlag=false, const std::vector< string > &keys=std::vector< string >\(\)\)](#)
- virtual void [column \(const string &colname, Column *value\)](#)
- int [numCols \(\) const](#)
- void [numCols \(int value\)](#)

8.37.1 Detailed Description

[Table](#) is the abstract common interface to Binary and Ascii [Table](#) HDUs.

[Table](#) is a subclass of [ExtHDU](#) that contains an associative array of [Column](#) objects. It implements methods for reading and writing columns

8.37.2 Constructor & Destructor Documentation

8.37.2.1 CCfits::Table::Table (const Table & right)

copy constructor.

8.37.2.2 CCfits::Table::~Table () [virtual]

destructor.

8.37.2.3 CCfits::Table::Table (FITSBase * *p*, HduType *xtype*, const string & *hduName*, int *rows*, const std::vector< string > & *columnName*, const std::vector< string > & *columnFmt*, const std::vector< string > & *columnUnit* = std::vector<string>(), int *version* = 1) [protected]

Constructor to be used for creating new HDUs.

Parameters:

p The [FITS](#) file in which to place the new [HDU](#)

rows The number of rows in the new [HDU](#) (the value of the NAXIS2 keyword).

columnType a vector of types for the columns.

columnName a vector of names for the columns.

columnFmt the format strings for the columns

columnUnit the units for the columns.

version a version number

8.37.2.4 CCfits::Table::Table (FITSBase * *p*, HduType *xtype*, const string & *hduName* = string("")), int *version* = 1) [protected]

Constructor to be called by operations that read [Table](#) specified by hduName and version.

Parameters:

xtype The type of the [Table](#) (AsciiTbl or BinaryTbl)

8.37.2.5 CCfits::Table::Table (FITSBase * *p*, HduType *xtype*, int *number*) [protected]

[Table](#) constructor for getting Tables by number.

Necessary since EXTNAME is a reserved not required keyword, and users may thus read [FITS](#) files without an extension name. Since an [HDU](#) is completely specified by extension number, this is part of the public interface.

8.37.3 Member Function Documentation**8.37.3.1 void CCfits::Table::column (const string & *colname*, Column * *value*) [protected, virtual]**

set the column with name colname to the input value.

Reimplemented from [CCfits::ExtHDU](#).

8.37.3.2 std::map< string, Column *> & CCfits::Table::column () [inline, virtual]

return a reference to the array containing the columns.

To be used in the implementation of subclasses.

8.37.3.3 Column & CCfits::Table::column (int colIndex) const [virtual]

return a reference to the column identified by colIndex.

Throws [NoSuchColumn](#) if the index is out of range -index must satisfy (1 <= index <= [numCols\(\)](#)).

N.B. the column number is assigned as 1-based, as in FORTRAN rather than 0-based as in C.

Exceptions:

Table::NoSuchColumn passes colIndex to the diagnostic message printed when the exception is thrown

Reimplemented from [CCfits::ExtHDU](#).

8.37.3.4 Column & CCfits::Table::column (const string & colName) const [virtual]

return a reference to the column of name colName.

Exceptions:

Table::NoSuchColumn passes colName to the diagnostic message printed when the exception is thrown

Reimplemented from [CCfits::ExtHDU](#).

8.37.3.5 const std::map< string, Column *> & CCfits::Table::column () const [inline]

return a reference to the array containing the columns.

This public version might be used to query the size of the column container in a routine that manipulates column table data.

8.37.3.6 void CCfits::Table::deleteColumn (const string & name) [virtual]

delete a column in a [Table](#) extension by name.

Parameters:

columnName The name of the column to be deleted.

Exceptions:

WrongExtensionType if extension is an image.

Reimplemented from [CCfits::ExtHDU](#).

8.37.3.7 void CCfits::Table::deleteRows (const std::vector< long > & *rowList*)

delete a set of rows in the table specified by an input array.

Parameters:

rowlist The vector of row numbers to be deleted.

Exceptions:

FitsError thrown if the underlying cfitsio call fails to return without error.

8.37.3.8 void CCfits::Table::deleteRows (long *first*, long *number* = 1)

delete a range of rows in a table.

In both this and the overloaded version which allows a selection of rows to be deleted, the cfitsio library is called first to perform the operation on the disk file, and then the [FITS](#) object is updated.

Parameters:

first the start row of the range

number the number of rows to delete; defaults to 1.

Exceptions:

FitsError thrown if the cfitsio call fails to return without error.

8.37.3.9 void CCfits::Table::init (bool *readFlag* = false, const std::vector< string > & *keys* = std::vector<string>()) [protected]

“Late Constructor.” wrap-up of calls needed to construct a table. Reads header information and sets up the array of column objects in the table.

Protected function, provided to allow the implementation of extensions of the library.

8.37.3.10 void CCfits::Table::insertRows (long *first*, long *number* = 1)

insert empty rows into the table.

Parameters:

first the start row of the range

number the number of rows to insert.

Exceptions:

FitsError thrown if the underlying cfitsio call fails to return without error.

8.37.3.11 void CCfits::Table::numCols (int *value*) [inline, protected]

set the number of Columns in the [Table](#).

8.37.3.12 int CCfits::Table::numCols () const [inline, protected]

return the number of Columns in the [Table](#) (the TFIELDS keyword).

8.37.3.13 void CCfits::Table::rows (long *numRows*) [inline]

set the number of rows in the [Table](#).

8.37.3.14 long CCfits::Table::rows () const [inline, virtual]

return the number of rows in the table (NAXIS2).

Reimplemented from [CCfits::ExtHDU](#).

8.37.3.15 void CCfits::Table::updateRows ()

update the number of rows in the table.

Called to force the [Table](#) to reset its internal "rows" attribute. public, but is called when needed internally.

The documentation for this class was generated from the following files:

- Table.h
- Column.h
- Table.cxx

8.38 CCfits::FITSUtil::MatchType::UnrecognizedType Class Template Reference

exception thrown by [MatchType](#) if it encounters data type incompatible with cfitsio.

```
#include <FITSUtil.h>
```

8.38.1 Detailed Description

template<typename T> class CCfits::FITSUtil::MatchType::UnrecognizedType

exception thrown by [MatchType](#) if it encounters data type incompatible with cfitsio.

The documentation for this class was generated from the following file:

- FITSUtil.h

8.39 CCfits::Column::WrongColumnType Class Reference

Exception thrown on attempting to access a scalar column as vector data.

```
#include <Column.h>
```

Inheritance diagram for CCfits::Column::WrongColumnType::

Public Methods

- [WrongColumnType](#) (const string &*diag*, bool *silent*=true)

8.39.1 Detailed Description

Exception thrown on attempting to access a scalar column as vector data.

This exception will be thrown if the user tries to call a read/write operation with a signature appropriate for a vector column on a scalar column, or vice versa. For example in the case of write operations, the vector versions require the specification of (a) a number of rows to write over, (b) a vector of lengths to write to each row or (c) a subset specification. The scalar versions only require a number of rows if the input array is a plain C-array, otherwise the range to be written is the size of the input vector.

8.39.2 Constructor & Destructor Documentation

8.39.2.1 CCfits::Column::WrongColumnType::WrongColumnType (const string &*diag*, bool *silent* = true)

Exception ctor, prefixes the string "[FitsError](#): Attempt to return scalar data from vector column, or vice versa - [Column](#): " before the specific message.

Parameters:

diag A specific diagnostic message, usually the column name.

silent if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

8.40 CCfits::ExtHDU::WrongExtensionType Class Reference

Exception to be thrown on unmatched extension types.

```
#include <ExtHDU.h>
```

Inheritance diagram for CCfits::ExtHDU::WrongExtensionType::

Public Methods

- [WrongExtensionType](#) (const string &msg, bool silent=true)

8.40.1 Detailed Description

Exception to be thrown on unmatched extension types.

This exception is to be thrown if the user requested a particular extension and it does not correspond to the expected type.

8.40.2 Constructor & Destructor Documentation

8.40.2.1 CCfits::ExtHDU::WrongExtensionType::WrongExtensionType (const string & msg, bool silent = true)

Exception ctor, prefixes the string "Fits Error: wrong extension type" before the specific message.

Parameters:

- msg* A specific diagnostic message
silent if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- ExtHDU.h
- ExtHDU.cxx

Index

~AsciiTable
 CCfits::AsciiTable, 40

~BinTable
 CCfits::BinTable, 47

~Column
 CCfits::Column, 53

~ExtHDU
 CCfits::ExtHDU, 68

~FITS
 CCfits::FITS, 81

~HDU
 CCfits::HDU, 93

~Keyword
 CCfits::Keyword, 107

~Table
 CCfits::Table, 126

~auto_array_ptr
 CCfits::FITSUtil::auto_array_ptr,
 43

addColumn
 CCfits::AsciiTable, 40
 CCfits::BinTable, 47
 CCfits::ExtHDU, 68

addImage
 CCfits::FITS, 81

addKey
 CCfits::HDU, 93

addNullValue
 CCfits::Column, 54

addTable
 CCfits::FITS, 82

AsciiTable
 CCfits::AsciiTable, 39, 40

auto_array_ptr
 CCfits::FITSUtil::auto_array_ptr,
 42

axes
 CCfits::HDU, 94

axis
 CCfits::HDU, 94

BinTable
 CCfits::BinTable, 45, 46

bitpix
 CCfits::HDU, 94

CantCreate
 CCfits::FITS::CantCreate, 49

CantOpen
 CCfits::FITS::CantOpen, 50

CCfits::AsciiTable, 37
 ~AsciiTable, 40
 addColumn, 40
 AsciiTable, 39, 40
 clone, 41
 readData, 41

CCfits::BinTable, 44
 ~BinTable, 47
 addColumn, 47
 BinTable, 45, 46
 clone, 47
 readData, 47

CCfits::Column, 51
 ~Column, 53
 addNullValue, 54
 Column, 53, 54
 comment, 54
 dimen, 55
 display, 55
 fitsPointer, 55
 format, 55
 index, 55
 isRead, 55
 makeHDUCurrent, 55
 name, 55
 put, 56
 read, 56, 57
 readArrays, 57
 readData, 58
 repeat, 58
 rows, 58
 scale, 58
 setDimen, 58
 setDisplay, 58
 type, 59

unit, 59
varLength, 59
width, 59
write, 59–63
writeArrays, 63
zero, 63
CCfits::Column::InsufficientElements,
 98
 InsufficientElements, 99
CCfits::Column::InvalidDataType, 100
 InvalidDataType, 100
CCfits::Column::InvalidNumberOf-
 Rows, 103
 InvalidNumberOfRows, 103
CCfits::Column::InvalidRowNumber,
 104
 InvalidRowNumber, 104
CCfits::Column::InvalidRow-
 Parameter, 105
 InvalidRowParameter, 105
CCfits::Column::NoNullValue, 112
 NoNullValue, 113
CCfits::Column::RangeError, 124
 RangeError, 124
CCfits::Column::WrongColumnType,
 131
 WrongColumnType, 132
CCfits::ExtHDU, 65
 ~ExtHDU, 68
 addColumn, 68
 clone, 69
 column, 69
 deleteColumn, 70
 ExtHDU, 68
 gcount, 70
 makeThisCurrent, 70
 name, 70, 71
 pcount, 71
 read, 71, 72
 readData, 72
 readHduName, 73
 rows, 73
 version, 73
 write, 73, 74
 xtension, 75
CCfits::ExtHDU::WrongExtension-
 Type, 132
 WrongExtensionType, 133
CCfits::FITS, 75
 ~FITS, 81
 addImage, 81
 addTable, 82
 clearErrors, 82
 clone, 82
 copy, 82
 currentExtension, 83
 currentExtensionName, 83
 deleteExtension, 83
 destroy, 83
 extension, 84
 filter, 84
 FITS, 77–80
 flush, 85
 name, 85
 pHDU, 85
 read, 85, 86
 resetPosition, 87
 setVerboseMode, 87
 verboseMode, 87
CCfits::FITS::CantCreate, 49
 CantCreate, 49
CCfits::FITS::CantOpen, 50
 CantOpen, 50
CCfits::FITS::NoSuchHDU, 115
 NoSuchHDU, 116
CCfits::FITS::OperationNot-
 Supported, 117
 OperationNotSupported, 118
CCfits::FitsError, 87
 FitsError, 88
CCfits::FitsException, 88
 FitsException, 90
CCfits::FitsFatal, 90
 FitsFatal, 90
CCfits::FITSUtil::auto_array_ptr, 41
 ~auto_array_ptr, 43
 auto_array_ptr, 42
 get, 43
 operator *, 43
 operator=, 43
 operator[], 43

release, 43
remove, 44
reset, 44
CCfits::FITSUtil::CAarray, 48
 operator(), 48
CCfits::FITSUtil::CVAarray, 64
 operator(), 64
CCfits::FITSUtil::CVarray, 64
 operator(), 65
CCfits::FITSUtil::MatchName, 109
CCfits::FITSUtil::MatchNum, 109
CCfits::FITSUtil::MatchPtrName, 110
CCfits::FITSUtil::MatchPtrNum, 110
CCfits::FITSUtil::MatchType, 111
CCfits::FITSUtil::Match-
 Type::UnrecognizedType,
 131
CCfits::HDU, 91
 ~HDU, 93
 addKey, 93
 axes, 94
 axis, 94
 bitpix, 94
 clone, 94
 comment, 95
 deleteKey, 95
 fitsPointer, 95
 getComments, 95
 getHistory, 95
 HDU, 93
 history, 95
 index, 95
 keyWord, 96
 makeThisCurrent, 96
 naxes, 96
 operator!=, 96
 operator==, 97
 parent, 97
 readAllKeys, 97
 readKey, 97
 readKeys, 97
 scale, 97
 setKeyWord, 97
 writeComment, 98
 writeDate, 98
 writeHistory, 98
 zero, 98
CCfits::HDU::InvalidExtensionType,
 101
 InvalidExtensionType, 101
CCfits::HDU::InvalidImageDataType,
 102
 InvalidImageDataType, 102
CCfits::HDU::NoNullValue, 111
 NoNullValue, 112
CCfits::HDU::NoSuchKeyword, 116
 NoSuchKeyword, 117
CCfits::Keyword, 106
 ~Keyword, 107
 clone, 107
 comment, 107
 fitsPointer, 107
 keytype, 107, 108
 Keyword, 107
 name, 108
 operator!=, 108
 operator=, 108
 operator==, 108
 parent, 108
 write, 108
CCfits::PHDU, 118
 clone, 120
 PHDU, 120
 read, 120–122
 scale, 122
 write, 122, 123
 zero, 123
CCfits::Table, 125
 ~Table, 126
 column, 127, 128
 deleteColumn, 128
 deleteRows, 129
 init, 129
 insertRows, 129
 numCols, 130
 rows, 130
 Table, 126, 127
 updateRows, 130
CCfits::Table::NoSuchColumn, 114
 NoSuchColumn, 114, 115
clearErrors
CCfits::FITS, 82

clone
 CCfits::AsciiTable, 41
 CCfits::BinTable, 47
 CCfits::ExtHDU, 69
 CCfits::FITS, 82
 CCfits::HDU, 94
 CCfits::Keyword, 107
 CCfits::PHDU, 120

Column
 CCfits::Column, 53, 54

column
 CCfits::ExtHDU, 69
 CCfits::Table, 127, 128

comment
 CCfits::Column, 54
 CCfits::HDU, 95
 CCfits::Keyword, 107

copy
 CCfits::FITS, 82

currentExtension
 CCfits::FITS, 83

currentExtensionName
 CCfits::FITS, 83

deleteColumn
 CCfits::ExtHDU, 70
 CCfits::Table, 128

deleteExtension
 CCfits::FITS, 83

deleteKey
 CCfits::HDU, 95

deleteRows
 CCfits::Table, 129

destroy
 CCfits::FITS, 83

dimen
 CCfits::Column, 55

display
 CCfits::Column, 55

extension
 CCfits::FITS, 84

ExtHDU
 CCfits::ExtHDU, 68

filter

 CCfits::FITS, 84

FITS
 CCfits::FITS, 77–80

FITS Exceptions, 36

FitsError
 CCfits::FitsError, 88

FitsException
 CCfits::FitsException, 90

FitsFatal
 CCfits::FitsFatal, 90

fitsPointer
 CCfits::Column, 55
 CCfits::HDU, 95
 CCfits::Keyword, 107

FITSUtil, 37

flush
 CCfits::FITS, 85

format
 CCfits::Column, 55

gcount
 CCfits::ExtHDU, 70

get
 CCfits::FITSUtil::auto_array_ptr,
 43

getComments
 CCfits::HDU, 95

getHistory
 CCfits::HDU, 95

HDU
 CCfits::HDU, 93

history
 CCfits::HDU, 95

index
 CCfits::Column, 55
 CCfits::HDU, 95

init
 CCfits::Table, 129

insertRows
 CCfits::Table, 129

InsufficientElements
 CCfits::Column::Insufficient-
 Elements, 99

InvalidDataType

CCfits::Column::InvalidData-
Type, 100
InvalidExtensionType
 CCfits::HDU::InvalidExtension-
 Type, 101
InvalidImageDataType
 CCfits::HDU::InvalidImageData-
 Type, 102
InvalidNumberOfRows
 CCfits::Column::InvalidNumber-
 OfRows, 103
InvalidRowNumber
 CCfits::Column::InvalidRow-
 Number, 104
InvalidRowParameter
 CCfits::Column::InvalidRow-
 Parameter, 105
isRead
 CCfits::Column, 55
keytype
 CCfits::Keyword, 107, 108
Keyword
 CCfits::Keyword, 107
keyWord
 CCfits::HDU, 96
makeHDUCurrent
 CCfits::Column, 55
makeThisCurrent
 CCfits::ExtHDU, 70
 CCfits::HDU, 96
name
 CCfits::Column, 55
 CCfits::ExtHDU, 70, 71
 CCfits::FITS, 85
 CCfits::Keyword, 108
naxes
 CCfits::HDU, 96
NoNullValue
 CCfits::Column::NoNullValue,
 113
 CCfits::HDU::NoNullValue, 112
NoSuchColumn
 CCfits::Table::NoSuchColumn,
 114, 115
NoSuchHDU
 CCfits::FITS::NoSuchHDU, 116
NoSuchKeyword
 CCfits::HDU::NoSuchKeyword,
 117
numCols
 CCfits::Table, 130
OperationNotSupported
 CCfits::FITS::OperationNot-
 Supported, 118
operator *
 CCfits::FITSUtil::auto_array_ptr,
 43
operator!=
 CCfits::HDU, 96
 CCfits::Keyword, 108
operator()
 CCfits::FITSUtil::CAarray, 48
 CCfits::FITSUtil::CVAarray, 64
 CCfits::FITSUtil::CVarray, 65
operator=

 CCfits::FITSUtil::auto_array_ptr,
 43
 CCfits::Keyword, 108
operator==
 CCfits::HDU, 97
 CCfits::Keyword, 108
operator[]
 CCfits::FITSUtil::auto_array_ptr,
 43
parent
 CCfits::HDU, 97
 CCfits::Keyword, 108
pcount
 CCfits::ExtHDU, 71
PHDU
 CCfits::PHDU, 120
pHDU
 CCfits::FITS, 85
put
 CCfits::Column, 56
RangeError
 CCfits::Column::RangeError,
 124

read
 CCfits::Column, 56, 57
 CCfits::ExtHDU, 71, 72
 CCfits::FITS, 85, 86
 CCfits::PHDU, 120–122
readAllKeys
 CCfits::HDU, 97
readArrays
 CCfits::Column, 57
readData
 CCfits::AsciiTable, 41
 CCfits::BinTable, 47
 CCfits::Column, 58
 CCfits::ExtHDU, 72
readHduName
 CCfits::ExtHDU, 73
readKey
 CCfits::HDU, 97
readKeys
 CCfits::HDU, 97
release
 CCfits::FITSUtil::auto_array_ptr,
 43
remove
 CCfits::FITSUtil::auto_array_ptr,
 44
repeat
 CCfits::Column, 58
reset
 CCfits::FITSUtil::auto_array_ptr,
 44
resetPosition
 CCfits::FITS, 87
rows
 CCfits::Column, 58
 CCfits::ExtHDU, 73
 CCfits::Table, 130
scale
 CCfits::Column, 58
 CCfits::HDU, 97
 CCfits::PHDU, 122
setDimen
 CCfits::Column, 58
setDisplay
 CCfits::Column, 58
setKeyWord
 CCfits::HDU, 97
setVerboseMode
 CCfits::FITS, 87
Table
 CCfits::Table, 126, 127
type
 CCfits::Column, 59
unit
 CCfits::Column, 59
updateRows
 CCfits::Table, 130
varLength
 CCfits::Column, 59
verboseMode
 CCfits::FITS, 87
version
 CCfits::ExtHDU, 73
width
 CCfits::Column, 59
write
 CCfits::Column, 59–63
 CCfits::ExtHDU, 73, 74
 CCfits::Keyword, 108
 CCfits::PHDU, 122, 123
writeArrays
 CCfits::Column, 63
writeComment
 CCfits::HDU, 98
writeDate
 CCfits::HDU, 98
writeHistory
 CCfits::HDU, 98
WrongColumnType
 CCfits::Column::WrongColumn-
 Type, 132
WrongExtensionType
 CCfits::ExtHDU::Wrong-
 ExtensionType, 133
xtension
 CCfits::ExtHDU, 75

zero

CCfits::Column, [63](#)

CCfits::HDU, [98](#)

CCfits::PHDU, [123](#)